The C4 model

for visualising software architecture

Simon Brown

Модель C4 для визуализации архитектуры программного обеспечения

Саймон Браун

Эта книга продается в http://leanpub.com/visualising-software-architecture

Эта версия была опубликована на 2022-05-28

Это книга от Leanpub . Leanpub расширяет возможности авторов и издателей с помощью процесса бережливого издательского производства. Бережливое издательство — это публикация незавершенной электронной книги с использованием облегченных инструментов и множества итераций для получения обратной связи от читателей, поворота до тех пор, пока у вас не будет подходящей книги, и наращивания популярности, как только вы это сделаете.

© 2015 - 2022 Саймон Браун

Твитните эту книгу!

Пожалуйста, помогите Саймону Брауну, распространив информацию об этой книге в Twitter!

Рекомендуемый хэштег для этой книги — #sa4d.

Узнайте, что другие люди говорят о книге, нажав на эту ссылку, чтобы найти этот хэштег в Twitter:

#sa4d

Для Кирсти, Мэтью и Оливера

Содержание

А. О книге 1

Б. Об авторе 3

В. Мы не в состоянии общаться. . . 4

В.А Что случилось с SSADM, RUP, UML и т.д.? 4

В.Б Облегченный подход 5

В.В Нарисуйте одну или несколько диаграмм 6

В.Г С чего начать? 7

В.Д Некоторые примеры . . 8

В.Е Общие проблемы . . 19

В.Ж Скрытые предположения диаграмм 20

Г. Общий словарный запас 21

Г.А Общие абстракции над общей нотацией 21

Г.Б Статическая структура . . . 22

Г.В Компоненты vs код? 26

Г.Г Модули и подсистемы? 30

Г.Д Микросервисы?. 31

Г.Е Бессерверный? 31

Г.Ж Платформы, фреймворки и библиотеки? 32

Г.З Создайте свой собственный общий словарный запас . . . 32

Д. Модель C4. 33

Д.А Иерархические карты вашего кода . . . 34

E. Уровень 1: Диаграмма системного контекста . . . 37

E.A Намерение 37

E.Б Структура..... 37

E.В Азы..... 38

СОДЕРЖАНИЕ

6.4 Взаимодействия 43

6.5 Мотивация 44

6.6 Аудитория 44

6.7 Обязательно или необязательно? 44

7. Уровень 2: Схема контейнера 45

7.1 Намерение 45

7.2 Структура 45

7.3 Элементы 47

7.4 Взаимодействия 50

7.5 Мотивация 51

7.6 Аудитория 52

7.7 Обязательно или необязательно? 52

8. Уровень 3: Схема компонентов 53

8.1 Намерение 53

8.2 Структура 53

8.3 Элементы 55

8.4 Взаимодействия 58

8.5 Мотивация 59

8.6 Аудитория 59

8.7 Обязательно или необязательно? 59

9. Уровень 4: Диаграммы на уровне кода 60

9.1 Намерение 60

9.2 Структура 60

9.3 Мотивация 62

9.4 Аудитория 62

9.5 Обязательно или необязательно? 62

10. Нотация 62

10.1 Титулы 62

10.2 Ключи и легенды 63

10.3 Элементы 64

10,4 Строки 69

10.5 Макет 71

10.6 Ориентация 73

10.7 Аббревиатуры 73

10.8 Атрибуты качества 74

СОДЕРЖАНИЕ

К.И Область действия диаграммы 68

К.К Слушайте вопросы 71

Л. Диаграммы должны отражать реальность 72

Л.А Разрыв между моделью и кодом . . . 72

Л.Б Детали технологии на диаграммах 74

Л.В Вы бы закодировали это таким образом? 77

М. Схемы развертывания 79

М.А Диаграммы сети и инфраструктуры . . . 79

М.Б Диаграммы развертывания . . . 80

М.В Схемы облачной архитектуры 82

Н. Другие диаграммы 84

Н.А Архитектурные видовые модели 84

Н.Б Ландшафт системы 88

Н.В Макеты пользовательского интерфейса и каркасы . . . 90

Н.Г Бизнес-процессы и документооборот . . . 90

Н.Д Доменная модель 91

Н.Е Продолжительность и поведение 91

Н.Ж И еще . . . 95

О. Приложение А: Система финансовых рисков 96

О.А Фон..... . 96

О.Б Функциональные требования 97

О.В Нефункциональные требования 97

1. О книге

Я окончил университет в 1996 году, в то время, когда CASE и инструменты моделирования были популярны и широко использовались. Я помню, как вскоре после начала своей профессиональной карьеры я посещал обучающий курс по унифицированному языку моделирования и инструментам SELECT. В ряде проектов, над которыми я работал, широко использовались такие инструменты, как SELECT и Rational Rose, для создания диаграмм и документирования

проектирования программных систем. С глючными пользовательскими интерфейсами и уродливыми диаграммами, инструментарий, возможно, не был блестящим в то время, но он все равно был очень полезен, если использовать его прагматично.

Я очень визуальный человек. Мне нравится иметь возможность визуализировать проблему, прежде чем пытаться найти решение. Опишите мне бизнес-процесс, и я набросаю его краткое изложение. Поговорите со мной о бизнес-проблеме, и я, скорее всего, нарисую высокоуровневую модель предметной области. Визуализация проблемы — это способ для меня задавать вопросы и выяснять, понял ли я, о чем вы говорите. Мне также нравится делать наброски решений проблем, опять же, потому что это отличный способ вынести все на всеобщее обозрение таким образом, чтобы другие люди могли быстро понять.

Но где-то в начале 2000-х годов что-то произошло, вероятно, в результате Манифеста гибкой разработки программного обеспечения, который был опубликован за несколько лет до этого.
То, как команды создавали программное обеспечение, начало меняться: такие вещи, как диаграммы и документация, были отброшены вместе с большим дизайном. Я помню, как несколько команд разработчиков программного обеспечения сокращали количество диаграмм и документации, которые они создавали. На самом деле, я часто был единственным человеком в команде, который действительно понимал UML достаточно хорошо, чтобы создавать диаграммы с его помощью.

Перенесемся в наши дни, и создание диаграмм архитектуры программного обеспечения кажется чем-то вроде утраченного искусства. В течение нескольких лет я провожу учебные курсы по архитектуре программного обеспечения, часть из которых представляет собой простую архитектурную ката, где группам людей предлагается спроектировать программное решение и нарисовать несколько архитектурных схем для его описания. Более 10 000 человек и 30+ стран спустя, у меня есть гигабайты анекдотических фотодоказательств - большинство диаграмм используют специальную нотацию "прямоугольники и линии" без четкой нотации или семантики. Проектирование программного обеспечения — это то, где должна быть сложность, а не коммуникация о ней.

Эта книга посвящена визуальной коммуникации и документированию архитектуры программного обеспечения. На протяжении многих лет я видел множество споров о том, является ли разработка программного обеспечения искусством, ремеслом или инженерной дисциплиной. Хотя я думаю, что это должна быть инженерная дисциплина, я считаю, что мы находимся на расстоянии нескольких лет от того, чтобы это стало реальностью. Так что, хотя эта книга этого не делает

представляет собой формализованный, стандартизированный метод коммуникации архитектуры программного обеспечения, он предоставляет коллекцию легких идей и методов, которые тысячи людей по всему миру находят полезными. В основе этого лежит моя «модель C4» для визуализации архитектуры программного обеспечения.

2. Об авторе

Я являюсь независимым консультантом по разработке программного обеспечения, специализирующимся на архитектуре программного обеспечения, в частности, на техническом лидерстве, коммуникации и легковесных, прагматичных подходах к архитектуре программного обеспечения. Помимо того, что я являюсь автором книги Software Architecture for Developers, я создатель модели архитектуры программного обеспечения C4 и создал Structurizr, который представляет собой набор инструментов, помогающих визуализировать, документировать и исследовать архитектуру программного обеспечения.

Я регулярно выступаю на конференциях, встречах и организациях по разработке программного обеспечения по всему миру; Выступление с основными докладами, презентациями и семинарами по архитектуре программного обеспечения. В 2013 году я выиграл награду SATURN 2013 «Архитектура на практике», спонсируемую IEEE Software, за свою презентацию о конфликте между agile и архитектурой. Я выступал на мероприятиях и/или у меня есть клиенты в более чем тридцати странах мира.

Вы можете найти мой веб-сайт по адресу simonbrown.je , а меня можно найти в Твиттере по адресу @simonbrown.

3. У нас не удается выйти на связь

Мы достигли интересного момента в индустрии разработки программного обеспечения. Глобально распределенные команды создают программные системы масштаба Интернета на всех видах языков программирования, архитектуры которых варьируются от монолитных систем до систем, состоящих из десятков микросервисов. Гибкие и бережливые подходы в настоящее время больше не рассматриваются как нишевые способы создания программного обеспечения, и даже самые традиционные организации стремятся к быстрой обратной связи с минимальными жизнеспособными продуктами, чтобы доказать свои идеи. Такие методы, как автоматизированное тестирование и непрерывная поставка, в сочетании с мощью облачных вычислений делают это реальностью и для организаций любого размера. Но чего-то все еще не хватает.

Попросите кого-нибудь из строительной отрасли визуально представить архитектуру здания, и вам, скорее всего, будут представлены планы участка, планы этажей, виды фасадов, виды поперечного сечения и подробные чертежи. В противоположность этому, попросите разработчика программного обеспечения рассказать об архитектуре программного обеспечения с помощью диаграмм, и вы, скорее всего, получите беспорядочную путаницу из блоков и линий.

Те архитектурные схемы, которые висят у вас на стене в офисе или в вики - отражают ли они систему, которая на самом деле строится, или это концептуальные абстракции, которые не имеют ничего общего со структурой кода? Чтобы диаграммы были полезными, они должны отражать реальность.

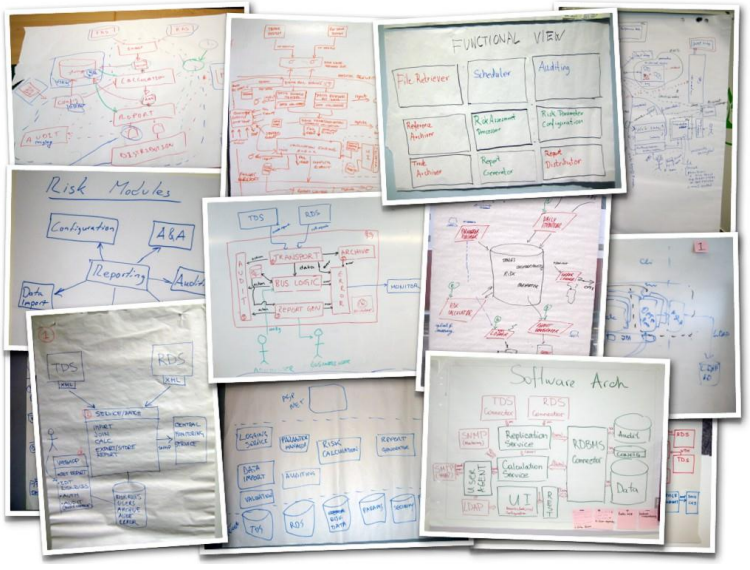
В ходе своих тренингов и семинаров я попросил тысячи разработчиков программного обеспечения по всему миру нарисовать диаграммы архитектуры программного обеспечения за последнее десятилетие и продолжаю делать это сегодня. Результаты до сих пор удивляют меня, наводя на мысль о том, что эффективная визуальная коммуникация архитектуры программного обеспечения — это навык, которого так не хватает в индустрии разработки программного обеспечения. Мы разучились визуализировать архитектуру программного обеспечения; как в процессе проектирования/разработки программного обеспечения, так и для долгосрочной документации.

3.1 Что случилось с SSADM, RUP, UML и т.д.?

Если вы вернетесь в прошлое, то увидите, что ряд структурированных процессов обеспечил точку отсчета для процесса проектирования программного обеспечения и того, как передать полученные проекты. Некоторые хорошо известные примеры включают рациональный унифицированный процесс (RUP) и структурированный

Метод системного анализа и проектирования (SSADM). Несмотря на то, что индустрия разработки программного обеспечения во многом продвинулась вперед, мы, кажется, забыли некоторые хорошие вещи, которые дали нам эти предыдущие подходы, в частности, связанные с некоторыми артефактами, которые эти подходы побуждали нас создавать.

Конечно, Unified Modelling Language (UML), стандартизированная нотация для передачи информации о проектировании программных систем, все еще жив. Однако, хотя вы можете спорить о том, предлагает ли UML эффективный способ коммуникации архитектуры программного обеспечения или нет, это часто не имеет значения, потому что многие команды уже отказались от UML или просто не знают об этом. Такие команды обычно предпочитают неформальные наброски в стиле коробок и линий, но часто эти диаграммы не имеют особого смысла, если они не сопровождаются подробным описанием.



Подборка типовых схем типа «коробки и линии»

Отказ от UML — это очень хорошо, но в гонке за гибкостью многие команды разработчиков программного обеспечения утратили способность к визуальному общению. Примеры эскизов архитектуры программного обеспечения (на фото) иллюстрируют типичные диаграммы архитектуры программного обеспечения, которые я вижу во время своих путешествий, и, как мы увидим в следующей главе, они страдают от ряда проблем.

3.2 Облегченный подход

Эта книга призвана решить эти проблемы, предлагая некоторые идеи и методы, которые команды разработчиков программного обеспечения могут использовать для визуализации и документирования своего программного обеспечения в упрощенном виде. Чтобы внести ясность, я не говорю о детальном моделировании, комплексных UML-моделях или разработке на основе моделей. Речь идет об эффективной и действенной передаче архитектуры программного обеспечения, которое вы создаете, с целью:

- Помогите каждому понять «общую картину» того, что строится, и как это вписывается в «большую картину» организации/среды, в которой она существует.
- Создайте общее видение для команды разработчиков.
- Предоставьте «карту», которую разработчики программного обеспечения могут использовать для навигации по исходному коду.
- Обеспечьте точку соприкосновения для технических обсуждений о новых функциях, техническом долге, анализе рисков и т. д.
- Ускорьте адаптацию новых разработчиков программного обеспечения в команду.
- Предоставьте возможность объяснить то, что создается, людям, не входящим в команду разработчиков, будь то технические или нетехнические специалисты.

Почему это важно? В современном мире гибкой поставки и бережливых стартапов многие команды разработчиков программного обеспечения утратили способность сообщать о том, что они создают, поэтому неудивительно, что этим же командам часто не хватает технического лидерства, направления и последовательности. Если вы хотите убедиться, что все вносят свой вклад в достижение одной и той же конечной цели, вы должны быть в состоянии эффективно донести видение того, что вы создаете. И если вам нужна гибкость и способность быстро передвигаться, вам также нужно уметь эффективно доносить это видение. Чтобы двигаться быстро, требуется хорошая коммуникация.

3.3 Нарисовать одну или несколько диаграмм

Как я уже упоминал во введении, я попросил тысячи разработчиков программного обеспечения нарисовать диаграммы архитектуры программного обеспечения во время семинаров, которые я проводил. Иногда это делается в рамках ката по архитектуре программного обеспечения, когда группам людей поручается проектирование программной системы. В других случаях это делается в рамках семинара по составлению диаграмм, где я прошу разработчиков программного обеспечения нарисовать несколько картинок, чтобы описать архитектуру системы, над которой они сейчас работают. В любом случае, результат один и тот же - специальный набор диаграмм "коробки и линии".

Задача буквально сформулирована как «нарисовать одну или несколько диаграмм архитектуры программного обеспечения для описания вашей программной системы». Как вы, наверное, догадываетесь, все результирующие диаграммы очень разные. Некоторые диаграммы демонстрируют очень высокий уровень абстракции, другие — низкоуровневые детали проектирования. Некоторые диаграммы показывают статичную структуру, другие показывают время выполнения и поведенческие аспекты. На некоторых диаграммах показаны варианты технологий, на большинстве — нет.

3.4 С чего начать?

Если задуматься, такой результат не вызывает удивления. Если спросить людей, что им показалось сложным в этом упражнении, то можно понять, что, возможно, визуальная коммуникация архитектуры программного обеспечения не является чем-то, чему учат заранее. На семинарах я регулярно слышу следующие вопросы:

- «Какие типы диаграмм мы должны нарисовать?»
- «Какую нотацию мы должны использовать?»
- «Какой уровень детализации мы должны представить?»
- «Кто является аудиторией для этих диаграмм?»

Я провожу это упражнение в группе, обычно от двух до пяти человек в группе. Вместо того, чтобы облегчить упражнение, наличие группы людей с разным образованием и опытом, как правило, усложняет ситуацию, поскольку время тратится впустую на обсуждение того, как лучше всего выполнить задачу. Это связано с тем, что, в отличие от строительной отрасли, в отрасли разработки

программного обеспечения отсутствует стандартный, последовательный способ осмысления, описания и визуальной передачи архитектуры программного обеспечения. Я считаю, что этому способствует ряд факторов:

А. В последние годы, спеша внедрить гибкие подходы, многие команды разработчиков программного обеспечения «выплеснули ребенка вместе с водой». Моделирование и документирование были отброшены вместе с традиционными процессами и методологиями, основанными на планах. Это может показаться немного экстремальным, но многие команды разработчиков программного обеспечения, с которыми я работаю, имеют очень ограниченное количество документации для своих программных систем.

Б. Команды, которые все еще видят ценность в документах и диаграммах, обычно отказываются от унифицированного языка моделирования (UML) в пользу более легкого и прагматичного подхода. Я расскажу о UML позже в этой книге, но мои анекдотические свидетельства, основанные на встречах и беседах с тысячами разработчиков программного обеспечения, говорят о том, что UML оптимистично используется лишь небольшим процентом разработчиков программного обеспечения.

В. Существует очень мало людей, которые учат команды разработчиков программного обеспечения эффективно моделировать, визуализировать и передавать архитектуру программного обеспечения. Основываясь на проведении небольшого количества семинаров для студентов бакалавриата в области компьютерных наук, это включает в себя и преподавателей в университетах.

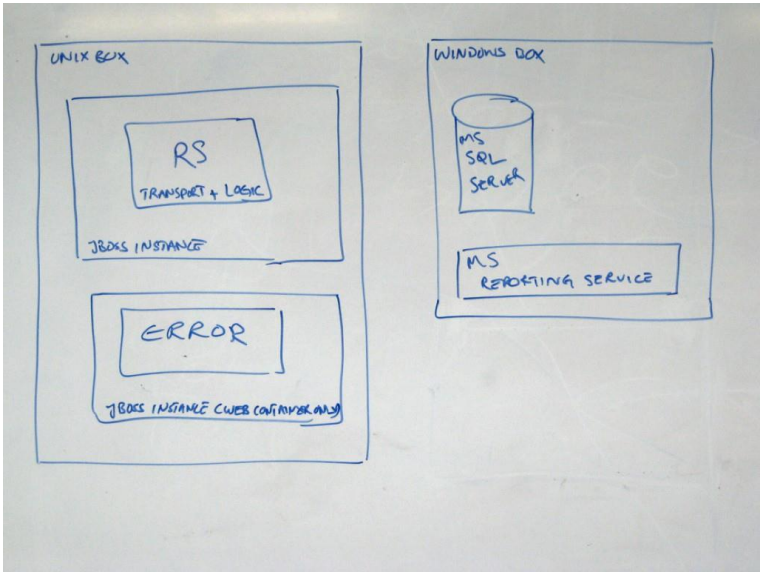
3.5 Некоторые примеры

Давайте рассмотрим несколько примеров. Приведенная ниже небольшая подборка фотографий взята с моих семинаров, где группам предлагалось разработать небольшую «систему финансовых рисков» для банка и нарисовать одну или несколько диаграмм, чтобы рассказать о ее программной архитектуре. Целью системы управления финансовыми рисками является импорт данных из двух источников данных («Система торговых данных» и «Система справочных данных»), объединение наборов данных, выполнение некоторых расчетов рисков и создание отчета, совместимого с Microsoft Excel, для ряда бизнес-пользователей. Подмножество этих бизнес-пользователей может дополнительно изменять некоторые

параметры, используемые во время расчетов. С полным набором требований к системе финансовых рисков вы можете ознакомиться в Приложении А.

Список покупок

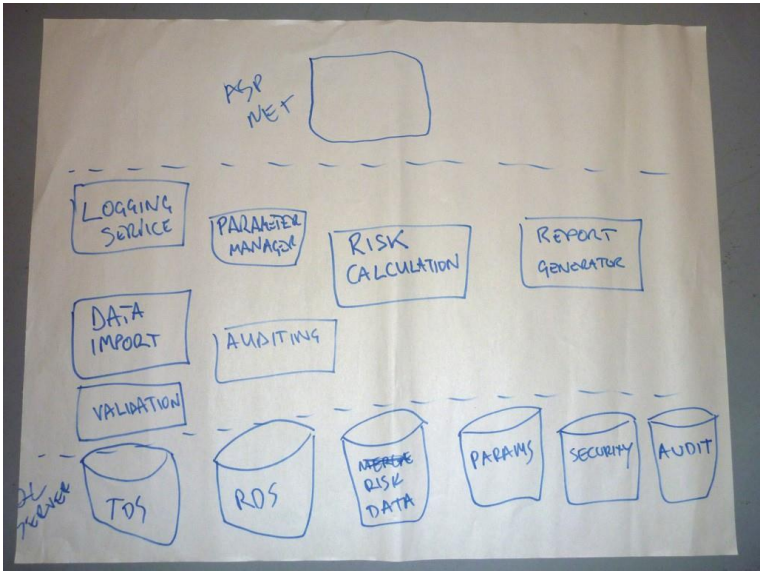
Независимо от того, является ли это единственной схемой архитектуры программного обеспечения или одной из коллекции диаграмм архитектуры программного обеспечения, эта диаграмма мало что говорит вам о решении. По сути, это просто список технологий.



Есть Unix-бокс и бокс Windows, с некоторыми дополнительными выборами продуктов, которые включают
JBoss (сервер приложений Java EE) и Microsoft SQL Server. Проблема в том, что я не знаю, что делают эти продукты, плюс, кажется, отсутствует связь между Unix box и Windows box. По сути, это маркированный список, который был представлен в виде диаграммы.

Коробки и никаких линий

Когда говорят об архитектуре программного обеспечения, они часто имеют в виду «коробки и линии». На следующей диаграмме есть прямоугольники, но нет линий.

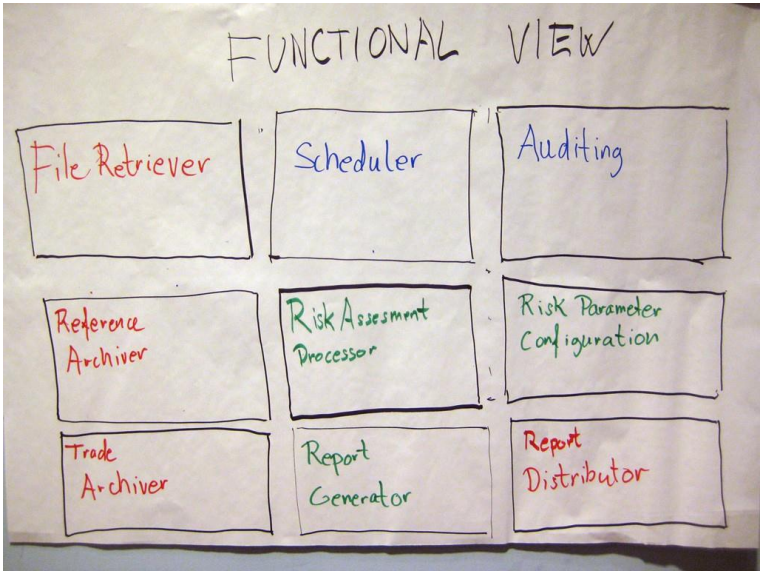


Это трехуровневое решение (я думаю), которое использует стек технологий Microsoft. Есть
ASP.NET веб-приложение вверху, которое, как я предполагаю, используется для какого-то взаимодействия с пользователем, хотя это не показано на диаграмме. Нижняя секция помечена как «SQL Server» и содержит множество отдельных цилиндров. Хотя, честно говоря, мне остается гадать, являются ли они отдельными серверами баз данных, схемами или таблицами.

Наконец, в середине находится набор блоков, которые, как я предполагаю, представляют собой компоненты, сервисы, модули и т.д. С одной стороны, здорово видеть, как средний уровень общего решения был разбит на более мелкие части, и это, безусловно, те типы компонентов/сервисов/модулей, которые я ожидал бы увидеть для такого решения. Но опять же, нет никаких обязанностей и взаимодействий. Архитектура программного обеспечения – это структура,
Речь идет о вещах (коробках) и о том, как они взаимодействуют (линии). На этой диаграмме есть одно, но нет другого. Это рассказ истории, но не всей истории.

«Функциональный вид»

Это похоже на предыдущую схему и по какой-то причине очень распространено, особенно в крупных организациях.

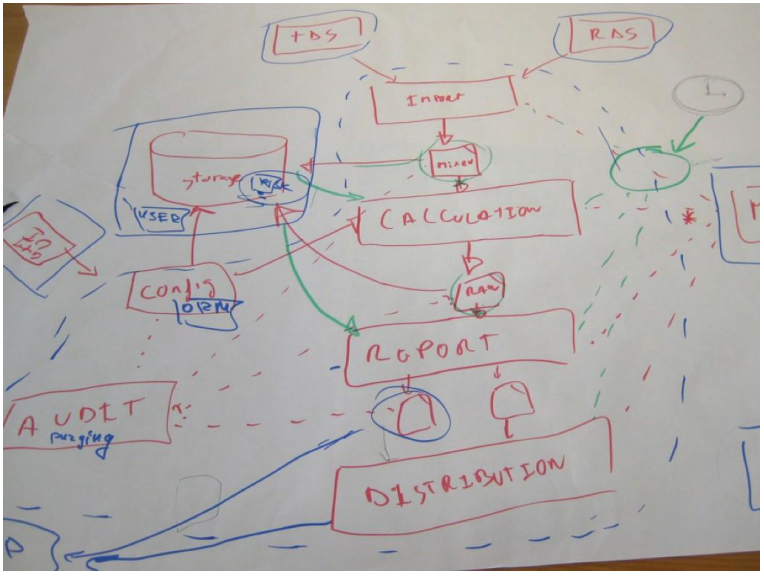


По сути, группа, создавшая эту диаграмму, просто задокументировала свое функциональное разложение решения на ряд более мелких элементов. Представьте себе архитектора, который рисует вам схему вашего нового дома, на котором просто есть набор коробок с надписями «Готовка», «Еда», «Сон», «Отдых» и т. д. или «Кухня», «Столовая», «Спальня», «Гостиная» и т. д.

Эта диаграмма страдает от той же проблемы, что и предыдущая диаграмма (нет обязанностей и взаимодействий), плюс у нас дополнительно есть цветовая кодировка для расшифровки. Можете ли вы понять, что означает цветовая кодировка? Связано ли это с функциями ввода и вывода? Или, может быть, это бизнес против инфраструктуры? Существующие или новые? Купить или построить? Или, может быть, у разных людей просто были ручки разного цвета! Кто знает. Меня часто спрашивают, почему центральное поле «Обработчик оценки рисков» имеет заметно более толстую рамку, чем другие блоки. Честно говоря, я не знаю, но подозреваю, что это просто потому, что маркер держали под другим углом.

Карта маршрутов авиакомпании

Это один из моих самых любимых фильмов. Это также была единственная диаграмма, которую эта конкретная группа использовала для представления своего решения.



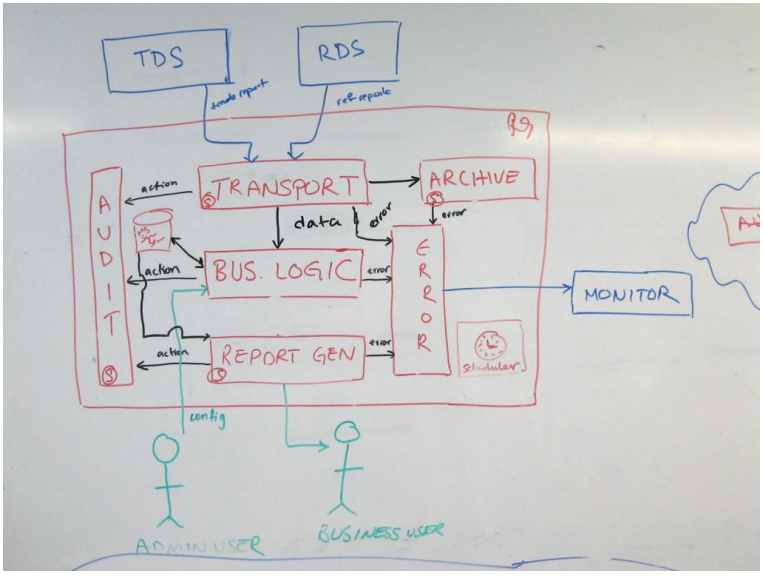
Центральный стержень этой диаграммы великолепен, потому что он показывает, как данные поступают из исходных систем данных (TDS и RDS), а затем проходят через ряд шагов для импорта данных, выполнения некоторых вычислений, создания отчетов и, наконец, их распространения. Это очень простая диаграмма активности, которая обеспечивает хороший высокоуровневый обзор того, что делает система. Но потом все идет не так.

Я думаю, что зеленый круг справа на диаграмме важен, потому что все указывает на него, но я не уверен, почему. И еще есть часы, которые, как я предполагаю, означают, что что-то запланировано на определенное время.

Левая часть диаграммы также запутанна, с различными линиями разных цветов и стилей, пересекающими друг друга. Если вы внимательно присмотритесь, то увидите буквы «UI» (пользовательский интерфейс) вверх ногами. В чем причина? Люди писали, сидя за столом.

В общем случае верно

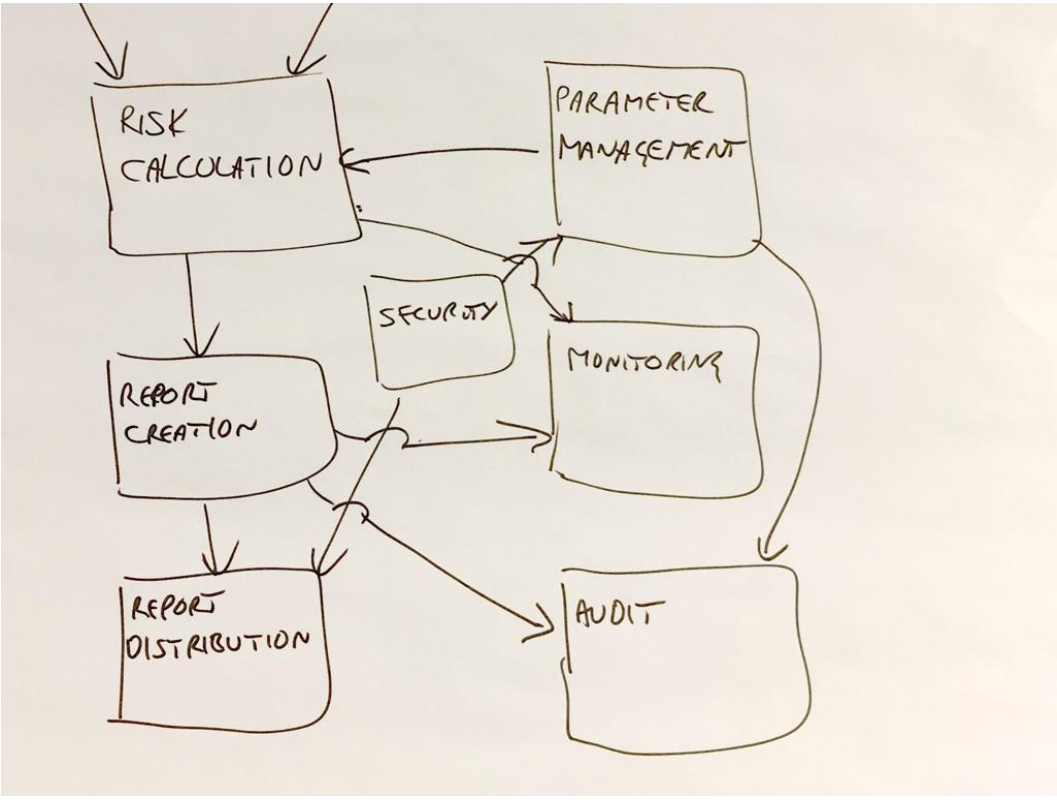
Это еще один очень распространенный стиль диаграммы. В следующий раз, когда кто-то попросит вас создать схему архитектуры программного обеспечения системы, подарите им эту фотографию, и все готово!



Это диаграмма в стиле "Архитектура программного обеспечения 101", где большая часть содержимого является общей. Игнорирование исходных систем данных в верхней части схемы (TDS и RDS); У нас есть поля с общими пометками «Транспорт», «Архив», «Аудит», «Генерация отчетов», «Обработка ошибок» и стрелки с пометками «Ошибка» и «Действие». И посмотрите на коробку в центре - она помечена как "бизнес-логика", что не очень информативно!

«Логический взгляд»

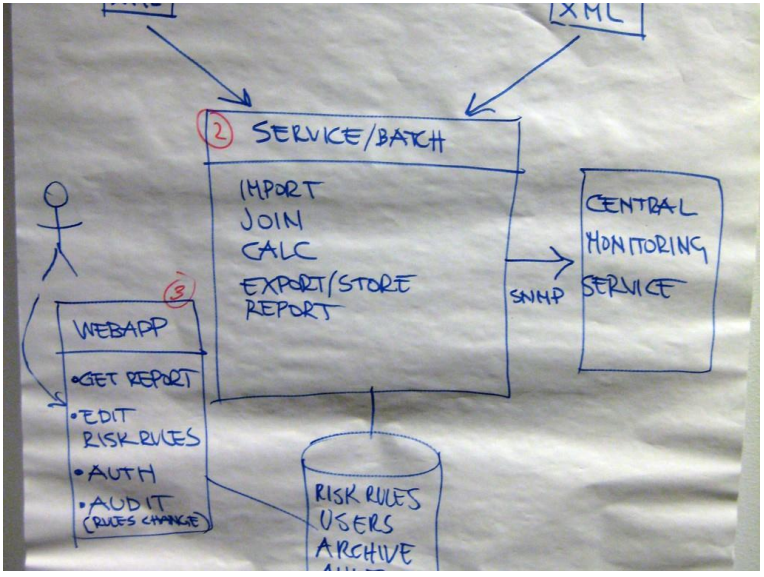
Эта диаграмма также относительно распространена. Он показывает логические (или концептуальные, или функциональные) строительные блоки, из которых состоит программная система, но предлагает очень мало другой информации.



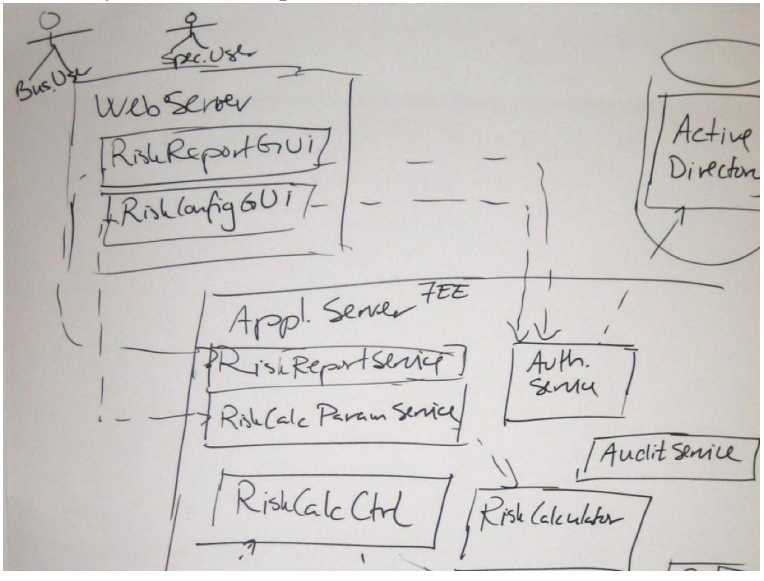
Существует распространенное заблуждение, что диаграммы «архитектуры программного обеспечения» должны быть «логическими» по своей природе, а не включать какие-либо ссылки на технологию или детали реализации, особенно до того, как будет написан какой-либо код. Мы рассмотрим это позже в книге.

Отсутствующие детали технологии

Эта диаграмма также относительно распространена. Он показывает общую форму архитектуры программного обеспечения (включая обязанности, которые мне очень нравятся), но выбор технологий остается на ваше воображение.



Аналогичным образом, следующая диаграмма говорит нам, что решение представляет собой n-уровневую систему Java EE, но, как и на предыдущей диаграмме, в ней опущены некоторые важные технологические детали.

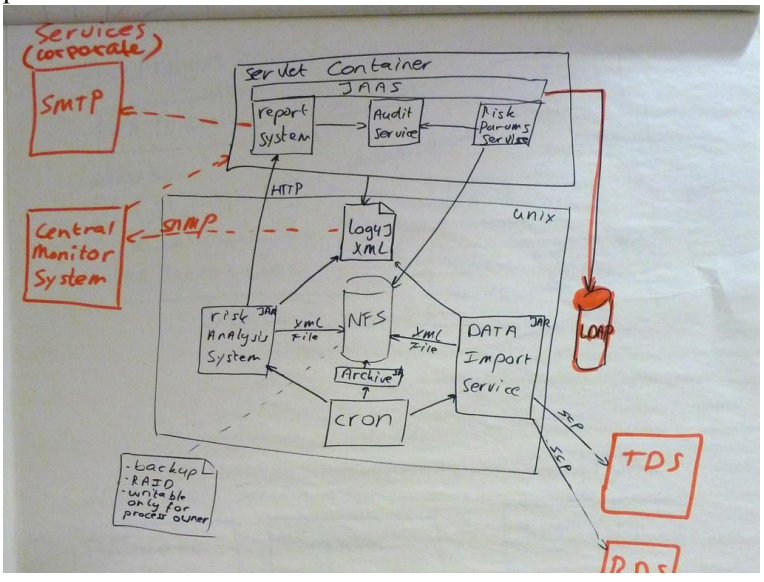


Линии между веб-сервером и сервером приложений не содержат информации о том, как происходит это взаимодействие. Это МЫЛО? Веб-API JSON? XML через HTTPS? Удаленный вызов метода? Асинхронный обмен сообщениями? Непонятно.

Мне часто говорят, что система финансовых рисков — это «простое решение, которое можно построить с помощью любой технологии», поэтому в любом случае это не имеет значения. Я не согласен с этим, и вопрос о включении или исключении выбора технологий более подробно рассматривается в другом месте книги.

Развертывание и контекст выполнения

Следующее решение представляет собой Java-решение, состоящее из веб-приложения и ряда серверных компонентов. Несмотря на то, что он предоставляет простой высокоуровневый обзор решения, в нем отсутствует некоторая информация, и вам нужно сделать несколько обоснованных предположений, чтобы заполнить пробелы.



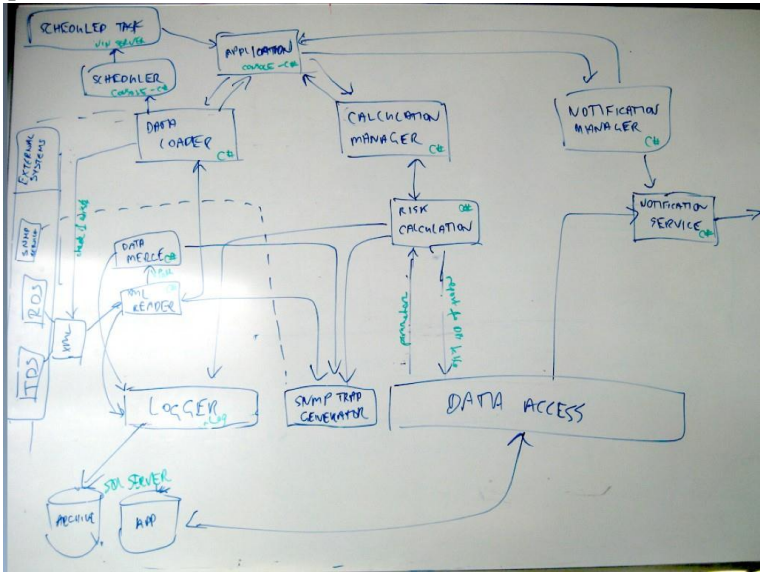
Если вы посмотрите на блок Unix в центре диаграммы, вы увидите два меньших блока с надписями
«Система анализа рисков» и «Сервис импорта данных». Если вы присмотритесь, то увидите, что оба поля имеют аннотацию "JAR", которая является механизмом развертывания Java-кода (Java ARchive). По сути, это ZIP-файл, содержащий

скомпилированный байт-код Java. Эквивалентом в мире .NET является библиотека DLL.

И в этом заключается двусмысленность. Что произойдет, если поместить JAR-файл на Unix-компьютер? Что ж, ответ не очень большой, кроме того, что он занимает некоторое место на диске. И cron (планировщик Unix) не выполняет JAR-файлы, если они не являются действительно автономными консольными приложениями, которые имеют метод "public static void main" в качестве точки входа в программу. Таким образом, я думаю, что оба этих JAR-файла на самом деле являются автономными приложениями, и это то, что я хотел бы видеть на диаграмме. Я хочу понять не механизм развертывания, а контекст выполнения.

Бездомный старый объект C# (HOCO)

Если вы слышали о "Plain Old C# Objects" (POCOs) или "Plain Old Java Objects" (POJOs), то это версия для бездомных. На этой диаграмме смешивается несколько различных уровней детализации.



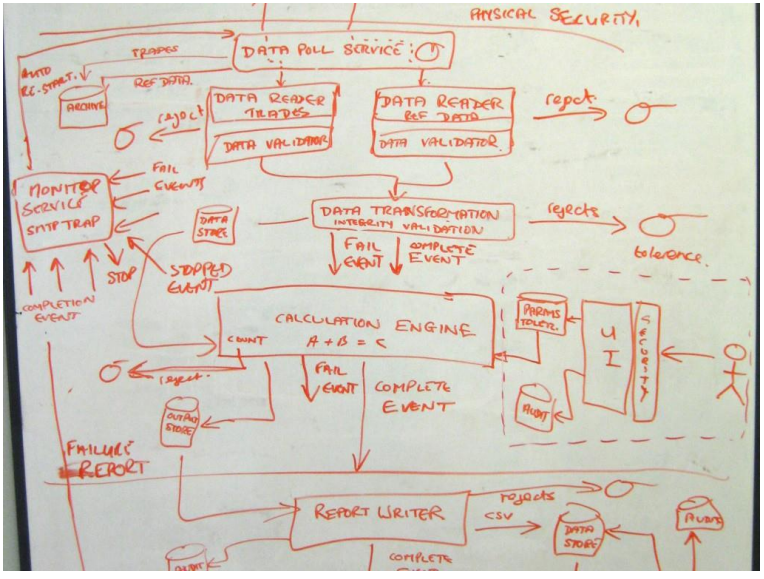
В левом нижнем углу диаграммы находится база данных SQL Server, а в левом верхнем углу диаграммы находится поле с надписью «Приложение». Обратите внимание, что тот же самый блок помечен (зеленым цветом) "ConsoleC#". По сути,

эта система состоит из консольного приложения C# и базы данных. Но как насчет других коробок?

Ну, большинство из них, похоже, являются компонентами, сервисами, модулями или объектами C#, и они очень похожи на то, что мы видели на некоторых других диаграммах. Есть также блок «доступ к данным» и блок «регистратор», которые могут быть фреймворками или архитектурными слоями. Все ли эти блоки представляют тот же уровень детализации, что и консольное приложение и база данных? Или они на самом деле являются частью приложения? Я подозреваю последнее, но отсутствие границ делает эту диаграмму запутанной. Я хотел бы нарисовать большую рамку вокруг большинства коробок, чтобы сказать: «Все эти вещи живут внутри консольного приложения». Я хочу дать этим коробкам дом. Опять же, я хочу понять, как система была разбита на более мелкие компоненты, но я также хочу знать и о контексте выполнения.

Выберите свое собственное приключение

Это средняя часть более сложной схемы.



Это немного похоже на те книги «выбери свое собственное приключение», которые я читал в детстве. Вы начинаете читать со страницы 1 и в конечном итоге приходите к развилке истории, где решаете, что должно произойти дальше. Если вы хотите атаковать большое страшное существо, с которым только что столкнулись, откройте

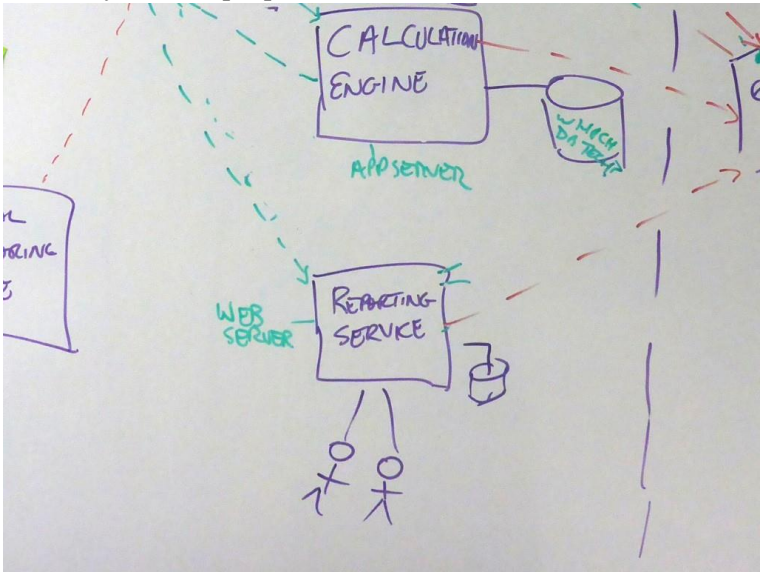
страницу 47. Если вы хотите убежать, как трус, то страница 205 для вас. Вы продолжаете делать похожий выбор, и в конце концов, к сожалению, ваш персонаж умирает, и вам приходится начинать все сначала.

На этой схеме то же самое. Вы начинаете с вершины и прокладываете свой путь вниз через сложный асинхронный и событийный стиль архитектуры. Часто приходится делать выбор - следует ли следовать "событию неудачи" или "завершенному событию"? Как и в книгах, все пути в конечном итоге ведут к ловушке (SNMP) в левой части диаграммы.

Диаграмма сложная, она пытается показать все, и один используемый цвет не помогает. Удаление некоторой информации и/или использование цветового кодирования для выделения различных путей через архитектуру было бы очень полезно.

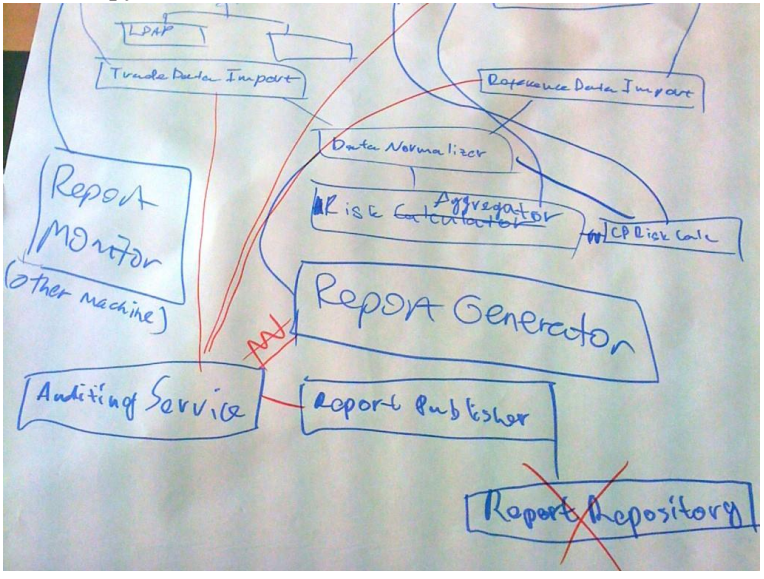
Штурмовики

Чтобы уловить что-то, что вы, возможно, заметили из предыдущих диаграмм, я регулярно вижу диаграммы, в которых фигурируют непомеченные пользователи/акторы. По сути, они являются безликими клонами. Я не знаю, кто они и зачем они используют это программное обеспечение.



Надо было использовать белую доску!

Итоговая диаграмма является отличным примером того, почему доски являются таким полезным оборудованием!



3.6 Распространенные проблемы

Шутки в сторону, эти диаграммы действительно страдают от одной или нескольких из следующих проблем:

- Обозначения (например, цветовое кодирование, формы и т. д.) не объясняются или являются непоследовательными.
- Назначение и значение элементов неоднозначны.
- Отношения между элементами отсутствуют или неоднозначны.
- Используются общие термины, такие как «бизнес-логика».
- Выбор технологии (или вариантов, если вы делаете предварительный дизайн) опущен.
- Уровни абстракции смешаны.
- Слишком много или слишком мало деталей.
- Никакого контекста или логической отправной точки.

Кроме того, проблемы, связанные с одной диаграммой, часто усугубляются при создании коллекции диаграмм:

- Нотация (цветовая кодировка, стили линий и т. д.) не согласуется между диаграммами.
- Именование элементов не согласуется между диаграммами.
- Логический порядок чтения диаграмм неясен.
- Четкого перехода между одной диаграммой и другой нет.

Примеры диаграмм типичны для того, что я вижу во время своих семинаров, и эти типы проблем невероятно распространены. Быстрый поиск изображений в Google обнаружит множество похожих блок-схем, которые страдают от многих из тех же проблем, которые мы уже видели. Я уверен, что вы видели подобные диаграммы и в своих организациях.

3.7 Скрытые предположения диаграмм

Один из самых простых способов понять, имеет ли диаграмма смысл, — это дать ее кому-то другому и попросить его интерпретировать ее без изложения. Я твердо верю, что диаграммы должны быть самостоятельными, во всяком случае, в какой-то степени. Любой нарратив должен дополнять диаграмму, а не объяснять ее. Тем не менее, я часто слышу группы в своем
На семинарах говорится следующее:

- «Мы поговорим по схемам».
- «Это не имеет смысла, но мы объясним это во время презентации».

Предположение о том, что диаграмма будет сопровождаться повествованием, создает разрыв между информацией, записанной на бумаге, и тем, что остается в головах людей. Диаграммы, требующие пояснения, имеют ограниченную ценность, особенно когда они используются для создания долговременной документации.

4. Общий словарный запас

Диаграммы, которые мы видели до сих пор, представляли собой специальный набор «коробок и линий». Несмотря на то, что нотация важна, одна из фундаментальных проблем, с которой, как мне кажется, мы сталкиваемся в индустрии разработки программного обеспечения, заключается в том, что нам не хватает общего, общего словаря, с помощью которого мы могли бы думать и описывать программные системы, которые мы создаем.

В следующий раз, когда вы будете участвовать в разговоре о разработке программного обеспечения, прислушайтесь к тому, как люди используют такие термины, как «компонент», «модуль», «подсистема» и т. д. Эти термины, как правило, неоднозначны. Например, словарное определение слова «компонент» — «часть большего целого». Представьте, что вы создаете веб-приложение, которое само использует базу данных. Учитывая словарное определение, оба следующих использования слова «компонент» являются допустимыми.

- «Веб-приложение является компонентом всей системы программного обеспечения».
- «Веб-приложение состоит из ряда компонентов».

По сути, слово «компонент» используется для описания двух очень разных уровней абстракции.

4.1 Общие абстракции над общей нотацией

Моя цель состоит в том, чтобы увидеть, как команды могут обсуждать структуру своих программных систем с помощью общего набора абстракций, а не пытаться понять, что пытаются показать различные элементы нотации. Хотя мне бы хотелось, чтобы индустрия разработки программного обеспечения создала стандартную нотацию, которую мы все понимаем (как, например, в электротехнической отрасли с электрическими схемами), я не думаю, что мы еще достигли этого. Возможно, общий набор абстракций более важен, чем общая нотация, учитывая нынешнюю незрелость и нехватку инженерной дисциплины в нашей отрасли.

Большинство карт являются отличным примером этого принципа в действии. Если вы возьмете две разные карты местности и расположите их рядом, они будут

отображать основные дороги, реки, озера, леса, города, районы, школы, церкви и так далее. Визуально, однако, эти карты, вероятно, будут использовать другую нотацию с точки зрения цветового кодирования, стилей линий, иконографии и т.д. Другими словами, карты показывают одни и те же вещи (одни и те же абстракции), но обозначения различаются. Ключ к их пониманию именно в этом; ключ или легенда, спрятанные где-нибудь в углу. То же самое мы можем сделать и с нашими диаграммами архитектуры программного обеспечения.

> Диаграммы — это карты, которые помогают разработчикам программного обеспечения ориентироваться в сложной кодовой базе.

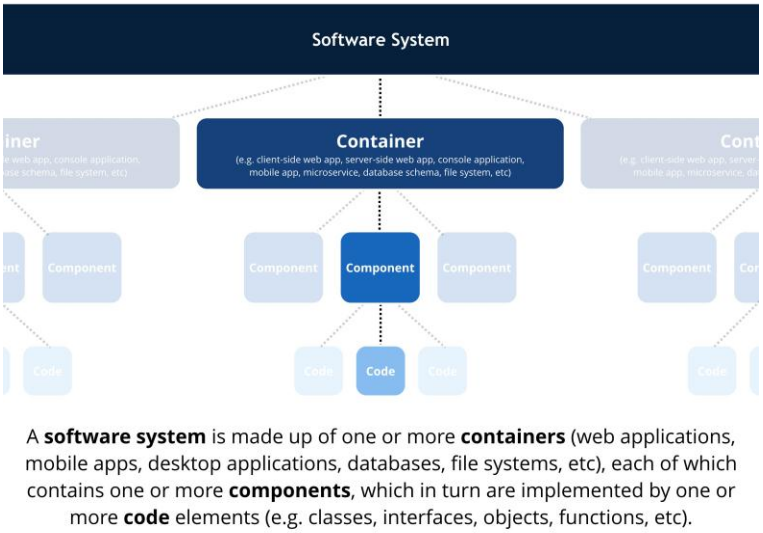
4.2 Статичная структура

Однако, чтобы добраться до этого момента, нам нужно договориться о некоторой лексике. И это тот шаг, который обычно пропускается во время начальной итерации моих семинаров по созданию диаграмм архитектуры программного обеспечения. Команды с головой погружаются в работу, не имея общего понимания терминов, которые они используют.

> «Это компонент нашей системы», — говорит один из разработчиков, указывая на коробку на диаграмме с надписью «Веб-приложение».

Я был свидетелем того, как группы людей обсуждали дизайн с использованием таких терминов, как «компонент», хотя они явно говорили об одном и том же. Тем не менее, все в группе не обращают на это внимания. Каждая группа должна договориться о лексике, терминологии и абстракциях, которые они собираются использовать. Затем нотация может развиваться.

Итак, если оставить в стороне нотацию (мы рассмотрим ее позже в книге), мой подход к решению этой проблемы заключается в том, чтобы ввести общий словарь, который мы можем использовать для описания нашего программного обеспечения. Основной аспект, который меня интересует, — это статичная структура. И меня интересует статическая структура с разных уровней абстракции. Как только эта статическая структура будет понята и начнет использоваться, ее легко дополнить другой информацией, чтобы проиллюстрировать характеристики времени выполнения/поведения, инфраструктуру, модели развертывания и т. д.



Простая модель архитектурных конструкций, используемая для определения статической структуры программной системы

Мне нравится думать о своей программной системе как об иерархии строительных блоков следующим образом:

> Программная система состоит из одного или нескольких контейнеров (веб-приложений, мобильных приложений, настольных приложений, баз данных, файловых систем и т.д.), каждый из которых содержит один или несколько компонентов, которые, в свою очередь, реализуются одним или несколькими элементами кода (например, классами, интерфейсами, объектами, функциями и т.д.). И люди используют программные системы, которые мы создаем.

Уровень 1: Программные системы

Программная система — это высший уровень абстракции, представляющий собой нечто, что приносит пользу своим пользователям, независимо от того, являются ли они людьми или нет.

Уровень 2: Контейнеры

Проще говоря, контейнер представляет собой нечто, в котором размещается код или данные, например приложение или хранилище данных. Контейнер — это то, что должно быть запущено для того, чтобы вся программная система работала. В реальном выражении контейнер – это что-то вроде:

- Серверное веб-приложение: веб-приложение Java EE, работающее на Apache Tomcat, приложение ASP.NET MVC, работающее на Microsoft IIS, приложение Ruby on Rails, работающее на WEBrick, приложение Node.js и т. д.
- Клиентское веб-приложение: JavaScript-приложение, работающее в веб-браузере с использованием AngularJS, Backbone.JS, jQuery и т. д.
- Настольное приложение на стороне клиента: настольное приложение Windows, написанное с использованием WPF, настольное приложение macOS, написанное с использованием Objective-C, кроссплатформенное настольное приложение, написанное с использованием JavaFX, и т. д.
- Мобильное приложение: приложение Apple iOS, приложение Android, приложение Microsoft Windows Phone и т. д.
- Консольное приложение на стороне сервера: автономное (например, "public static void main") приложение, пакетный процесс и т.д.
- Микросервис: Один микросервис, размещенный на чем угодно, от традиционного веб-сервера до чего-то вроде Spring Boot, Dropwizard и т. д.
- Бессерверная функция: Одна бессерверная функция (например, Amazon Lambda, Azure Function и т. д.).
- База данных: схема или база данных в системе управления реляционными базами данных, хранилище документов, графовая база данных и т. д., такие как MySQL, Microsoft SQL Server, Oracle Database, MongoDB, Riak, Cassandra, Neo4j и т. д.
- Blob или хранилище содержимого: хранилище BLOB-объектов (например, Amazon S3, Microsoft Azure Blob Storage и т. д.) или сеть доставки содержимого (например, Akamai, Amazon CloudFront и т. д.).
- Файловая система: Полная локальная файловая система или часть более крупной сетевой файловой системы (например,
  SAN, NAS и т.д.).
- Shell-скрипт: Один shell-скрипт, написанный на Bash и т.д.
- и так далее

Контейнер — это, по сути, контекст или граница, внутри которой выполняется какой-либо код или хранятся некоторые данные. Название «контейнер» было выбрано, потому что я хотел получить имя, которое не подразумевало бы ничего о физической природе того, как этот контейнер выполняется[1]. Например, некоторые веб-серверы запускают несколько потоков в одном процессе, в то время как другие запускают один поток в нескольких процессах. Когда я размышляю о статической структуре программной системы, я не хочу беспокоиться о том, использует ли веб-приложение один или несколько процессов операционной системы при обслуживании запросов. Это важная деталь, но
Мы можем вернуться к этому позже.

Контейнеры развертываются отдельно

Также стоит отметить, что каждый контейнер должен быть отдельно развертываемой вещью. Физическое развертывание — еще одна важная деталь, которую мы рассмотрим позже, но, во всяком случае, теоретически каждый контейнер может быть развернут или запущен на отдельном участке инфраструктуры;
Будь то физическая, виртуальная или контейнерная инфраструктура. Из этого следует, что взаимодействие между контейнерами, скорее всего, потребует внепроцессного или удаленного вызова процедуры через границу процесса и/или сети.

В качестве примера давайте представим, что вы создаете веб-сайт, состоящий из двух разных веб-приложений (например, настольной версии и мобильной версии, или версии для конечного пользователя, обслуживающей HTML, и конечной точки API, обслуживающей JSON). Существует несколько сценариев, которые следует рассмотреть:

А. Каждое веб-приложение упаковано в отдельно развертываемые единицы (например, два файла Java WAR, два ASP.NET веб-приложения и т. д.). Это два контейнера, независимо от того, развернуты ли оба развертываемых модуля на одном и том же физическом веб-сервере (оптимизация развертывания).
Б. Несмотря на то, что эти два веб-приложения логически разделены, на самом деле они неразделимы, поскольку упакованы как единая единица развертывания

[1] Я очень ценю то, что термин «контейнер» сейчас широко используется благодаря контейнеризации и таким технологиям, как Docker. Не стесняйтесь использовать что-то вроде "контекст времени выполнения", "среда выполнения" или "развертываемая единица", если вы предпочитаете избегать термина "контейнер" при обсуждении архитектуры программного обеспечения.

(например, один файл Java WAR или ASP.NET веб-приложение). Это одиночный контейнер.

То же самое верно и для схем реляционных баз данных. Я бы рассматривал две отдельные схемы как два отдельных контейнера, независимо от того, развернуты они на одном и том же сервере базы данных или нет.

В заключение скажу, что контейнер относится к контексту выполнения, и это действительно конструкция времени выполнения. Это означает, что библиотеки или модули (файлы e.g. JAR, DLL-файлы, сборки .NET и т. д.) не должны рассматриваться как контейнеры, если они не могут запускаться сами по себе, как, например, приложение Java или Spring Boot, упакованное в исполняемый JAR-файл.

Уровень 3: Компоненты

Слово «компонент» является чрезвычайно перегруженным термином в индустрии разработки программного обеспечения, но мне нравится думать о компоненте как о группе связанных функций, инкапсулированных за четко определенным интерфейсом. В модели C4 компоненты не являются отдельно развертываемыми единицами. Вместо этого контейнер является развертываемым блоком. Другими словами, все компоненты внутри контейнера обычно выполняются в одном и том же пространстве процесса.

Такие аспекты, как то, как эти компоненты упакованы (например, один компонент против многих компонентов в JAR-файле, DLL, общая библиотека и т. д.), являются ортогональными и, с моей точки зрения, не влияют на то, как мы думаем о компонентах.

Уровень 4: Код

Наконец, компоненты состоят из одного или нескольких элементов кода, созданных с использованием основных строительных блоков используемого языка программирования; классы, интерфейсы, перечисления, функции, объекты и т.д.

4.3 Компоненты vs код?

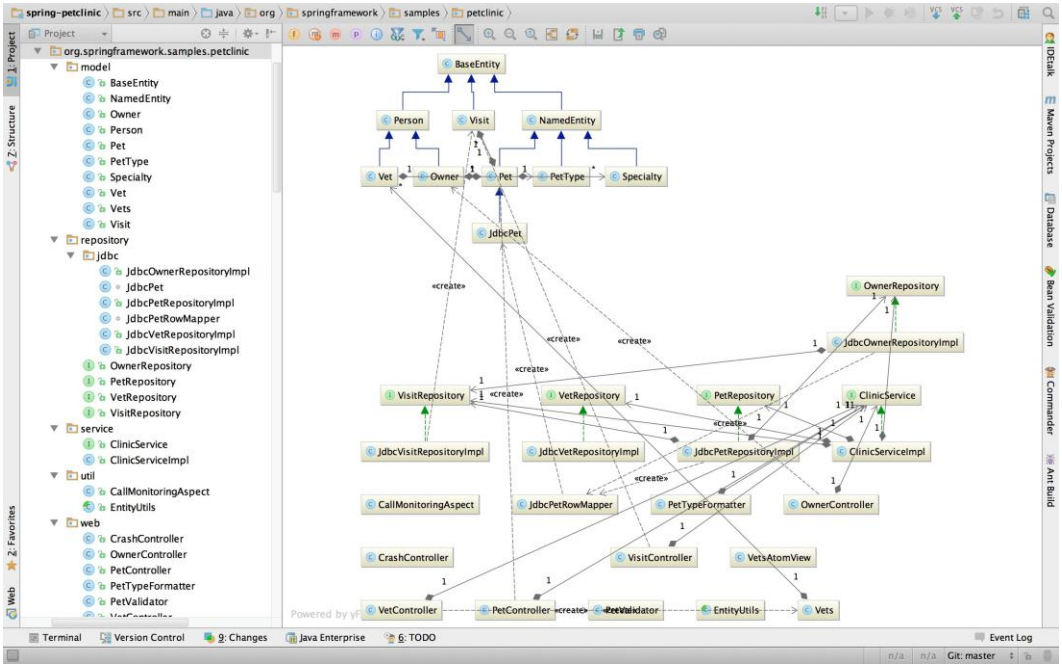
Компонент — это способ выйти на один уровень абстракции из стандартных блоков уровня кода, которые есть в используемой технологии. Например:

- Объектно-ориентированные языки программирования (например, Java, C#, C++ и т.д.): компонент состоит из классов и интерфейсов.
- Процедурные языки программирования (например, C): компонент может состоять из нескольких файлов C в определенном каталоге.
- JavaScript: Компонентом может быть модуль JavaScript, который состоит из ряда объектов и функций.
- Функциональные языки программирования: Компонент может быть модулем (концепция, поддерживаемая такими языками, как F#, Haskell и т.д.), который представляет собой логическую группировку связанных функций, типов и т.д.
- Реляционная база данных: компонент может представлять собой логическую группировку функциональных возможностей; на основе ряда таблиц, представлений, хранимых процедур, функций, триггеров и т.д.

Если вы используете объектно-ориентированный язык программирования, ваши компоненты будут реализованы с помощью одного или нескольких классов. Давайте рассмотрим быстрый пример, чтобы лучше определить, что такое компонент в контексте некоторого кода.

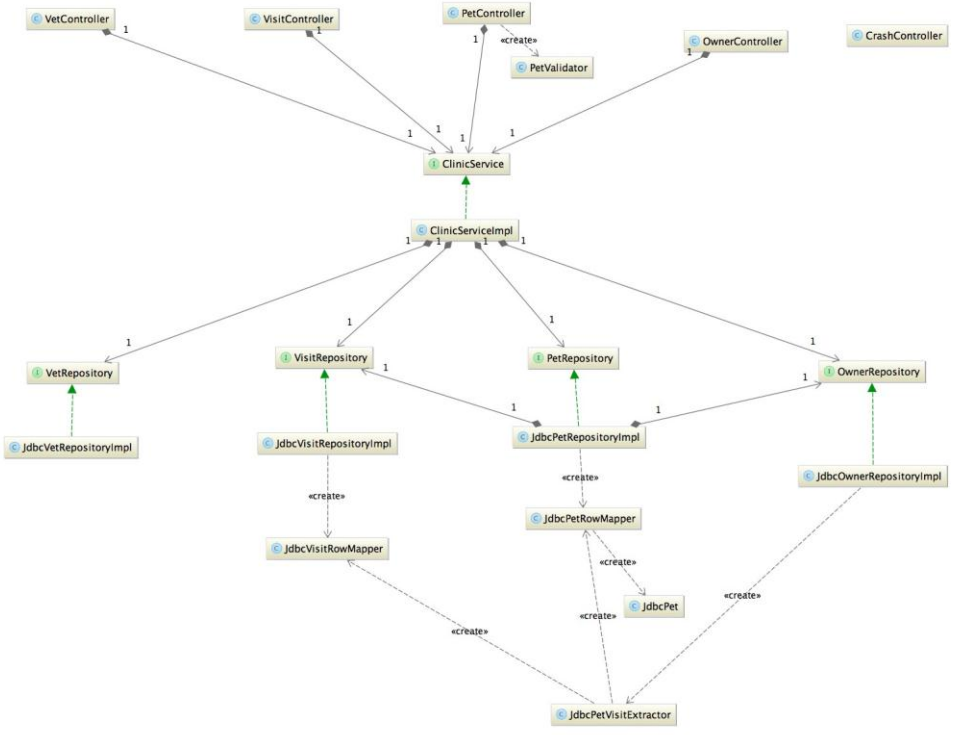
Приложение Spring PetClinic — это пример кодовой базы, иллюстрирующий, как создать веб-приложение Java с помощью платформы Spring MVC. С нетехнической точки зрения, это программная система, разработанная для воображаемой клиники для домашних животных, которая хранит информацию о домашних животных и их владельцах, посещениях клиники и ветеринарах, которые там работают. Система предназначена только для использования сотрудниками клиники. С технической точки зрения система Spring PetClinic состоит из веб-приложения и реляционной базы данных.

Версия веб-приложения, которую мы здесь рассмотрим, представляет собой типичную многоуровневую архитектуру, состоящую из ряда веб-контроллеров MVC, сервиса, содержащего «бизнес-логику», и нескольких репозиториев для доступа к данным. Также есть несколько классов предметной области и утилиты. Если вы скачаете копию репозитория GitHub, откроете ее в выбранной вами IDE и визуализируете ее, реконструируя диаграмму классов UML из кода, вы получите что-то вроде этого.

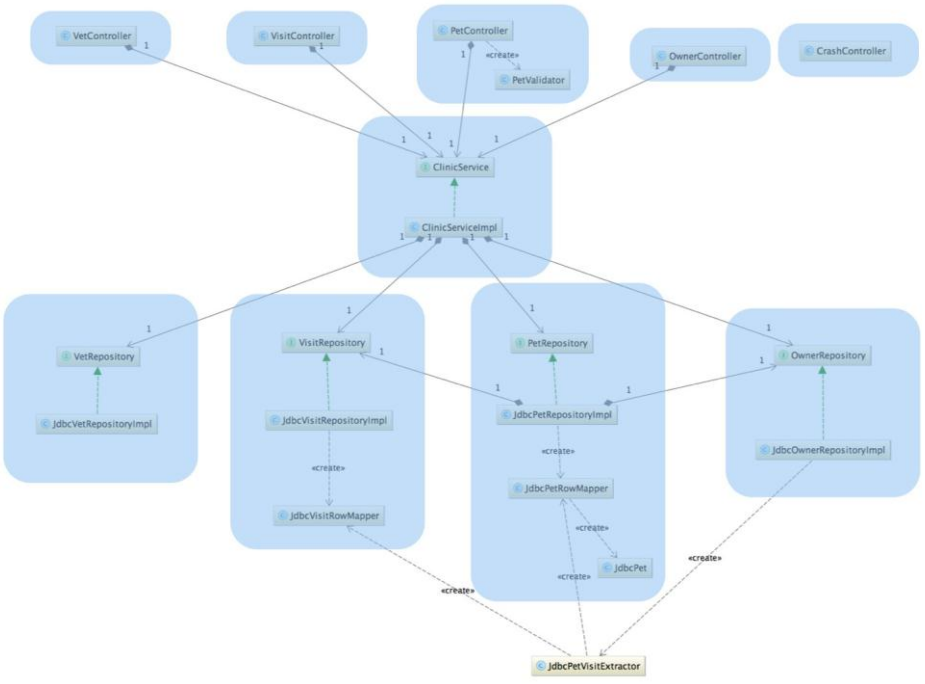


Как и следовало ожидать, на этой диаграмме показаны все классы Java и интерфейсы, составляющие веб-приложение Spring PetClinic, а также все отношения между ними. Свойства и методы скрыты на диаграмме, потому что они добавляют слишком много шума на картинку. Это не сложная кодовая база, но, показывая классы и интерфейсы, диаграмма показывает слишком много деталей.

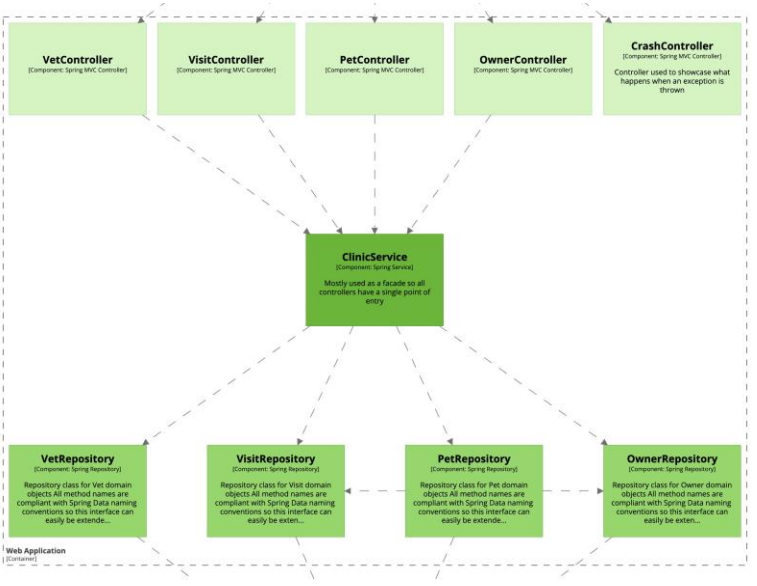
Давайте уберем те классы, которые бесполезны для обсуждения «архитектуры» системы. Другими словами, давайте покажем только те классы/интерфейсы, которые имеют некоторое значение с точки зрения статической структуры. Если говорить конкретно, то для данной конкретной кодовой базы это означает исключение классов model (domain) и util.



После небольшой перестановки у нас теперь есть более простая схема, с помощью которой можно рассуждать об архитектуре программного обеспечения. Мы также можем снова увидеть архитектурные слои (контроллеры, сервисы и репозитории). Но на этой диаграмме по-прежнему отображаются элементы уровня кода (т.е. классы и интерфейсы). Чтобы увеличить масштаб на один уровень, нам нужно определить, какие из этих элементов уровня кода могут быть сгруппированы вместе, чтобы сформировать «компоненты». Стратегия группировки элементов уровня кода в компоненты будет варьироваться от кодовой базы к кодовой базе (и мы обсудим это далее в книге), но для этой кодовой базы стратегия может выглядеть следующим образом.



Каждый из синих прямоугольников представляет то, что я считаю «компонентом» в этой кодовой базе. Подводя итог, можно сказать, что каждый из веб-контроллеров является отдельным компонентом, вместе с результатом объединения остальных интерфейсов и классов их реализации. Если убрать шум на уровне кода, то получится вот такая картина.



По сути, мы группируем классы и интерфейсы в компоненты, которые являются единицей связанной функциональности. Скорее всего, у вас будет общий код (например, абстрактные базовые классы, вспомогательные классы, вспомогательные классы и т.д.), который используется во многих компонентах, таких как `JdbcPetVisitExtractor` в этом примере. Некоторые из них могут быть подвергнуты рефакторингу и перемещены «внутри» того или иного компонента, но некоторые из них неизбежны.

Хотя этот пример иллюстрирует традиционную многоуровневую архитектуру, одни и те же принципы применимы независимо от того, как вы упаковываете свой код (например, по слоям, функциям или компонентам) или используемого архитектурного стиля (например, многоуровневый, шестиугольный, порты и адаптеры и т. д.). Моя цель во всем этом состоит в том, чтобы свести к минимуму и, по сути, устранить разрыв между тем, как разработчики программного обеспечения думают о компонентах с логической и физической точки зрения. Компоненты должны быть реальными вещами, очевидными в коде, а не логическими конструкциями, которые используются только в обсуждениях архитектуры.

4.4 Модули и подсистемы?

Если вы знакомы с определением архитектуры программного обеспечения из таких книг, как Программное обеспечение
Архитектура на практике, вы могли заметить, что я не использую термин «модуль» как часть определения статической структуры. Модуль обычно относится к единице реализации (например, библиотеке или некоторой другой коллекции программных элементов), которая может быть объединена с другими модулями в компонент, который сам создается для создания экземпляров компонента во время выполнения. Хотя эта модель имеет смысл, я считаю, что она добавляет дополнительный уровень детализации, который обычно не нужен, когда думаешь о программной системе с точки зрения «общей картины». По этой причине я намеренно избегаю использования термина «модуль» и вместо этого сосредотачиваюсь на идентификации более крупнозернистых компонентов в статической структуре.

Я также избегал использования термина «подсистема», который некоторые люди используют для обозначения набора связанных компонентов или функционального среза программной системы. Проблема, с которой я столкнулся с термином «подсистема», заключается в том, что часто бывает трудно сопоставить эту концепцию с реальной кодовой базой. Если концепция компонентов и модулей или систем и подсистем полезна, то не стесняйтесь встраивать ее в общий словарь, который вы создаете.

4.5 Микросервисы?

Учитывая степень ажиотажа и дискуссий вокруг микросервисов в настоящее время, стоит четко указать, как описывать микросервисы, используя терминологию, которую мы определили до сих пор. Вообще говоря, есть два варианта.

1. Микросервисы как программные комплексы

Если ваша программная система зависит от ряда микросервисов, которые находятся вне вашего контроля (например, они принадлежат и/или управляются отдельной командой), я бы рассматривал эти микросервисы как внешние программные системы, которые вы не можете видеть внутри.

2. Микросервисы в виде контейнеров

С другой стороны, если микросервисы являются частью создаваемой вами программной системы
(т.е. вы являетесь их владельцем), я бы рассматривал их как контейнеры, как и любые хранилища данных, которые используют эти микросервисы (это отдельные контейнеры). Точно так же, как модульное монолитное приложение представляет собой контейнер с несколькими компонентами, работающими внутри него, микросервис — это контейнер с (меньшим) количеством компонентов, работающих внутри него. Фактическое количество компонентов будет зависеть от стратегии реализации. Она может варьироваться от очень простой (т.е. такой, где микросервис представляет собой контейнер с одним компонентом, работающим внутри) до чего-то вроде мини-многоуровневой или шестиугольной архитектуры.

4.6 Бессерверный?

Я склонен относиться к бессерверным концепциям (например, Amazon Lambdas, Azure Functions) так же, как к микросервисам. Если вы создаете программную систему, состоящую из ряда бессерверных функций, думайте о них как о контейнерах, поскольку все они могут развертываться отдельно.

4.7 Платформы, фреймворки и библиотеки?

Вы также можете задаться вопросом, какое место во всем этом занимают платформы, фреймворки и библиотеки. В конце концов, платформы и фреймворки — это то, на чем вы создаете свое программное обеспечение, в то время как библиотеки — это то, что использует ваше программное обеспечение. В большинстве случаев это просто выбор технологий, которые используются компонентами, и поэтому они являются деталями реализации, а не компонентами сами по себе. Например, компоненты в веб-приложении Spring PetClinic используют фреймворк Spring, но я бы не стал показывать фреймворк Spring в качестве компонента. То же самое верно для библиотеки логирования, драйвера базы данных и т.д.

В большинстве случаев хороший дизайн[1] побудит вас обернуть компоненты из платформ, фреймворков и библиотек в свои собственные компоненты. Конечно, бывают случаи, когда ваше программное обеспечение может использовать компоненты, предоставляемые платформами, фреймворками и библиотеками. Понимание того, как вы используете такие компоненты, является ключом к пониманию того, как они вписываются в статическую модель вашего программного обеспечения.

4.8 Создание собственного общего словаря

Я проиллюстрировал лексику, которую использую здесь, и она работает для большинства организаций, с которыми я работаю. Но, конечно, универсальных правил не существует. Иногда, вместо того чтобы внедрять что-то новое, организации проще придерживаться словарного запаса, который они уже используют, гарантируя, что он четко определен и понятен всем.

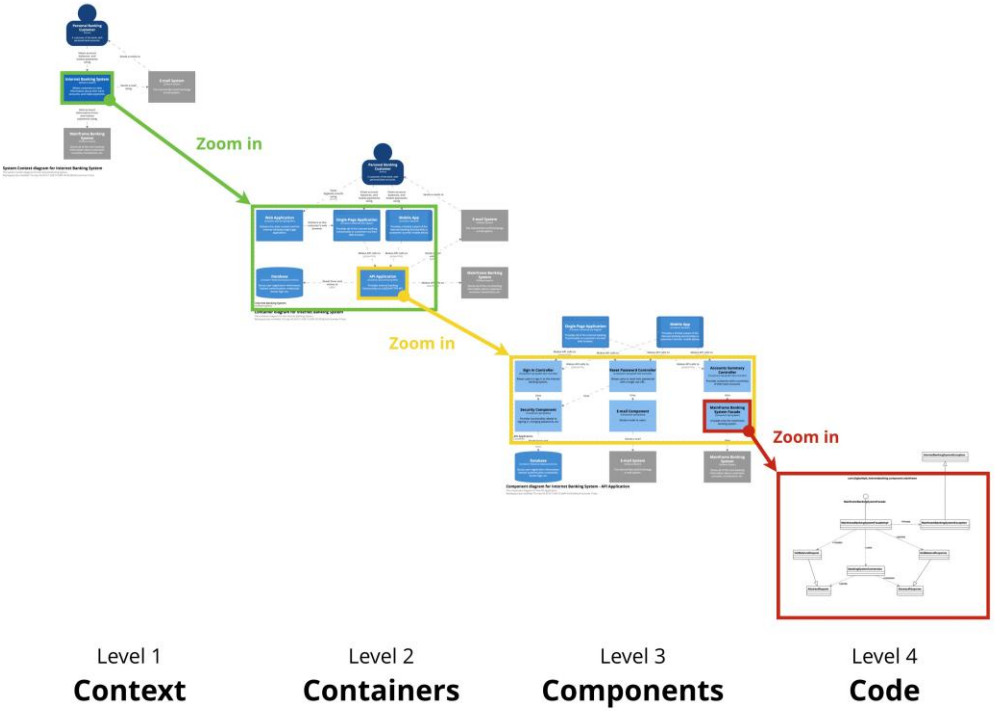
Например, одна организация, с которой я работал, создает программное обеспечение на C для работы на мобильных устройствах. Вместо терминов «контейнер» и «компонент» они используют термины «компонент» и «модуль» соответственно. В данном случае «компонент» относится к исполняемому файлу, встроенному в C, который, в свою очередь, состоит из нескольких модулей. Несмотря на то, что терминология отличается, у нас все еще есть иерархическая структура, которая может быть использована для описания программной системы на нескольких различных уровнях абстракции.

[1] Многие архитекторы программного обеспечения часто пытаются отделить свой код от кода, предоставляемого платформами, фреймворками и библиотеками, чтобы при необходимости можно было отложить принятие решений и изменить технологии.

5. Модель C4

Имея в виду общий словарь, мы можем теперь перейти к рисованию некоторых диаграмм на различных уровнях абстракции, чтобы визуализировать статическую структуру программной системы. Я называю это "моделью C4"; (Система) Контекст, контейнеры, компоненты и код.

А. Системный контекст: Диаграмма системного контекста служит отправной точкой, показывающей, как программная система вписывается в окружающий мир.
Б. Контейнеры: Диаграмма контейнеров позволяет масштабировать программную систему в области действия, показывая высокоуровневые технические строительные блоки (контейнеры) и то, как они взаимодействуют.
В. Компоненты: Диаграмма компонентов масштабируется до отдельного контейнера, показывая компоненты внутри него.
Г. Код: Диаграмма кода (например, класса UML) может быть использована для увеличения масштаба отдельного компонента, показывающего, как этот компонент реализован.

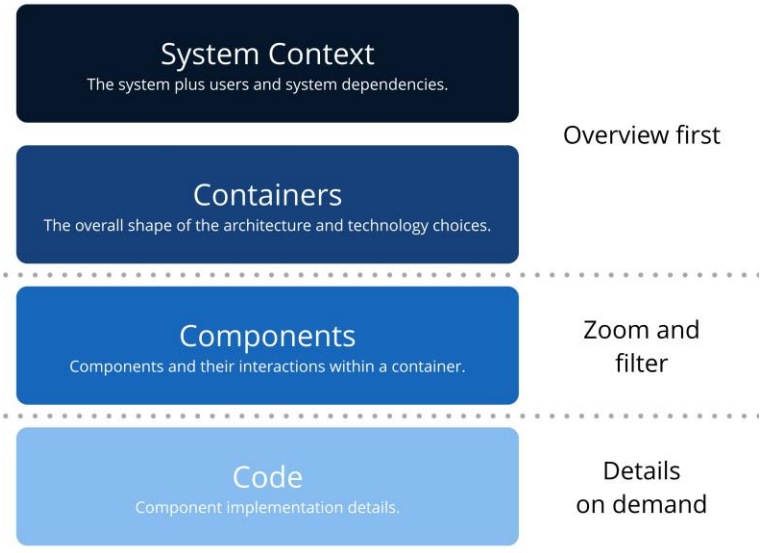


Краткое описание модели C4

5.1 Иерархические карты вашего кода

Диаграммы C4 можно рассматривать как набор карт для программной системы, который предоставляет вам возможность увеличивать и уменьшать масштаб с различными уровнями детализации. Мантра Шнейдермана — это простая концепция для понимания и визуализации больших объемов данных, но она очень хорошо подходит для модели C4, поскольку является иерархической.

Сначала обзор, масштабирование и фильтрация, а затем сведения по запросу



Мантра Шнейдермана

Системный контекст и контейнеры: сначала обзор

Моей отправной точкой для понимания любой программной системы является построение диаграммы системного контекста. Это помогает мне понять область применения системы, кто ее использует и каковы ключевые системные зависимости. Обычно он быстро рисуется и быстро понимается.

Далее я открою систему и нарисую диаграмму, показывающую контейнеры (веб-приложения, мобильные приложения, автономные приложения, базы данных, файловые системы, шины сообщений и т. д.), составляющие систему. Это показывает общую форму программной системы, то, как были распределены обязанности и какие ключевые технологические решения были сделаны.

Компоненты: масштабирование и фильтр

Как разработчикам, нам часто требуется больше деталей, поэтому я по очереди увеличиваю масштаб каждого (интересного) контейнера и показываю «компоненты» внутри него. Здесь я показываю, как каждое приложение было разложено на компоненты, а также кратко рассказываю о ключевых обязанностях и выборе

технологий для этих компонентов. Рисование диаграмм от руки может стать утомительным, поэтому в идеале вам следует обратить внимание на инструменты, которые помогут автоматизировать это.

Код: Подробности по запросу

При желании я могу продвинуться дальше по иерархии, чтобы показать элементы уровня кода (например, классы, интерфейсы, объекты, функции и т. д.), которые составляют определенный компонент. В конечном счете, однако, эта деталь находится в коде, и мы, как разработчики программного обеспечения, можем получить ее по запросу через наши IDE.

Разные диаграммы, разные истории

В следующий раз, когда вас попросят создать несколько диаграмм архитектуры программного обеспечения (будь то понимание существующей системы, представление обзора системы или археология программного обеспечения), я советую помнить мантру Шнейдермана. Начните с самого верха и работайте над деталями, создавая историю, которая углубляется в детали по мере ее продвижения. Различные уровни диаграмм позволяют рассказывать разные истории для разных аудиторий; Некоторые из них будут техническими специалистами, некоторые нет.

Замечу, что модель C4 — это не описание процесса проектирования, а просто набор диаграмм, которые можно использовать для описания статической структуры программной системы. Тем не менее, и мы рассмотрим это позже, хотя модель C4 описывает диаграммы, охватывающие четыре уровня абстракции, вам не обязательно создавать каждую диаграмму на каждом уровне. Я рекомендую всем командам создавать диаграммы системного контекста и контейнеров и действительно задумываться о том, дают ли диаграммы компонентов и кода достаточную выгоду, учитывая стоимость их создания и обновления.

6. Уровень 1: Диаграмма контекста системы

Контекстная диаграмма системы может быть полезной отправной точкой для создания диаграмм и документирования программной системы, позволяя вам сделать шаг назад и взглянуть на общую картину.

6.1 Намерение

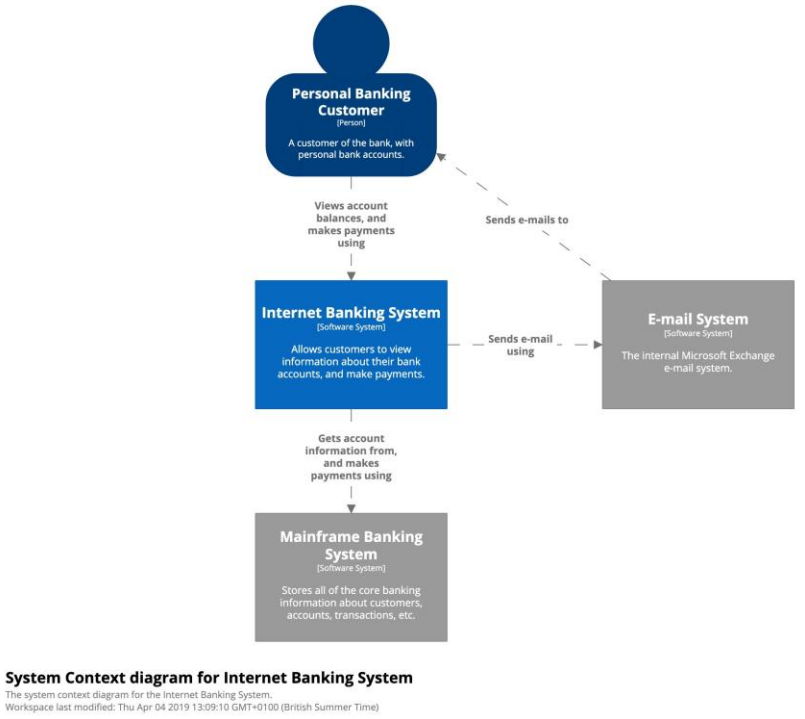
Диаграмма системного контекста поможет вам ответить на следующие вопросы.

А. Какую программную систему мы создаем (или создали)?
Б. Кто его использует?
В. Как он вписывается в существующую среду?

6.2 Структура

Нарисуйте блок-схему, показывающую вашу программную систему в виде коробки в центре, окруженной ее пользователями и другими программными системами, с которыми она взаимодействует. Детали здесь не важны, так как это увеличенное изображение, показывающее общую картину программной системы и непосредственного мира вокруг нее. Основное внимание должно быть сосредоточено на людях (актерах, ролях, персонажах и т. д.) и программных системах, а не на технологиях, протоколах и других низкоуровневых деталях. Это своего рода диаграмма, которую вы можете показать нетехническим людям.

Давайте рассмотрим на примере. Это диаграмма системного контекста для вымышленной системы интернет-банкинга. Он показывает людей, которые его используют, и другие программные системы, с которыми система интернет-банкинга взаимосвязана.



Контекстная схема системы Интернет-банкинг

Таким образом, Частные клиенты банка используют систему «Интернет-банкинг» для просмотра информации о своих банковских счетах, а также для осуществления платежей. Сама система интернет-банкинга использует для этого существующую банковскую систему Mainframe Banking и использует существующую систему электронной почты банка для отправки электронных писем клиентам.

6.3 Элементы

Диаграмма системного контекста обычно включает в себя два типа элементов; люди и программные системы.

Народ

Это люди, которые используют вашу программную систему. Будете ли вы моделировать их как отдельных людей, пользователей, ролей, актеров или персонажей — это ваш выбор. Как правило, я собираю следующую информацию о людях:

- Имя: имя человека, пользователя, роли, актера или персонажа.
- Описание: Краткое описание человека, его роли, обязанностей и т.д.

Программные комплексы

Это другие программные системы, с которыми взаимодействует ваша программная система. Как правило, эти программные системы находятся за пределами вашей собственной программной системы, и вы не несете ответственности за них и не являетесь их владельцем. Пример с интернет-банкингом в этом отношении очень ясен. Как команда, создающая систему интернет-банкинга, мы не владеем и не несем никакой ответственности за существующую банковскую систему основного компьютера или систему электронной почты. По этой причине они включены в схему, чтобы проиллюстрировать, что они являются зависимостями системы интернет-банкинга, а не тем, что мы строим сами.

Опять же, я соберу следующую информацию о каждой программной системе:

- Name: Название программного комплекса.
- Описание: Краткое описание программного комплекса, его обязанностей и т.д.

Граница предприятия

По желанию, я могу захотеть получить некоторую информацию о местонахождении людей и/или программных систем относительно моей точки отсчета. Если я создаю программную систему внутри организации/предприятия, эта программная система может взаимодействовать с людьми и программными системами за пределами этой границы. В этом слегка измененном примере пунктирная линия представляет собой границу банка и используется для иллюстрации того, что находится внутри, а что находится за пределами банка.

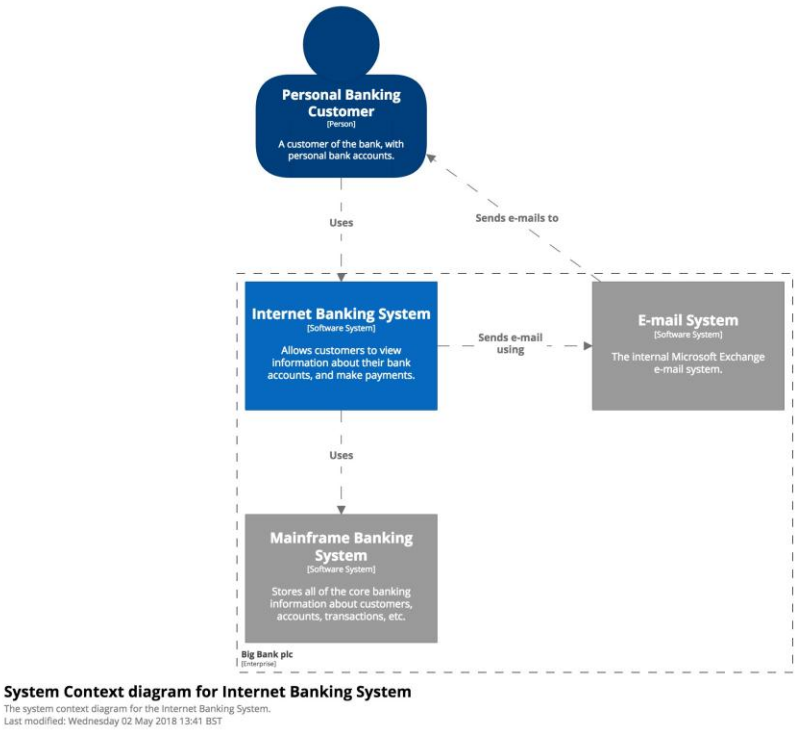


Диаграмма системного контекста для системы Интернет-банкинга, дополнительно показывающая границы предприятия

6.4 Взаимодействия

Постарайтесь аннотировать каждое взаимодействие между элементами на диаграмме некоторой информацией о цели этого взаимодействия. Это позволяет избежать создания диаграммы, на которой набор блоков каким-то образом связан набором неоднозначных линий. Опять же, постарайтесь сохранить этот относительно высокий уровень и не думайте, что вам нужно включать множество технических деталей (например, протоколы, форматы данных и т.д.).

6.5 Мотивация

Вы можете спросить, в чем смысл такой простой диаграммы. Вот почему это полезно:

- Он делает контекст и область действия программной системы явными, так что нет никаких предположений.
- Он показывает, что добавляется (с высокого уровня) в существующую среду.
- Это высокоуровневая диаграмма, которую технические и нетехнические специалисты могут использовать в качестве отправной точки для обсуждения.
- Он обеспечивает отправную точку для определения того, с кем вам потенциально нужно поговорить с пониманием межсистемных интерфейсов.

Диаграмма системного контекста не показывает много деталей, но она помогает создать сцену и является отправной точкой для других диаграмм. Я часто рисую эту диаграмму во время семинара по сбору требований, чтобы убедиться, что все понимают масштаб того, что нам было поручено создать. Это может быть отличным инструментом для сбора и анализа требований.

6.6 Аудитория

Технические и нетехнические специалисты, как внутри, так и за пределами непосредственной команды разработчиков программного обеспечения.

6.7 Обязательно или необязательно?

Все программные системы должны иметь диаграмму системного контекста.

7. Уровень 2: Схема контейнера

Как только вы поймете, как ваша программная система вписывается в общую среду с помощью
Диаграмма системного контекста, полезным следующим шагом является иллюстрация высокоуровневых технологических решений с помощью диаграммы контейнера.

7.1 Намерение

На диаграмме контейнера показана высокоуровневая форма архитектуры программного обеспечения и распределение обязанностей между ней. В нем также показаны основные технологические решения, способы их использования и взаимодействие контейнеров друг с другом. Это высокоуровневая технологическая диаграмма, полезная как для разработчиков программного обеспечения, так и для сотрудников службы поддержки/эксплуатации. Схема контейнера помогает ответить на следующие вопросы:

А. Какова общая форма программного комплекса?
Б. Каковы высокоуровневые технологические решения?
В. Как распределяются обязанности по системе?
Г. Как контейнеры взаимодействуют друг с другом?
Д. Где мне как разработчику нужно писать код для реализации функций?

7.2 Структура

Нарисуйте блок-схему, показывающую высокоуровневые технические элементы (контейнеры), из которых состоит ваша программная система. Это пример схемы контейнера для вымышленной системы интернет-банкинга.

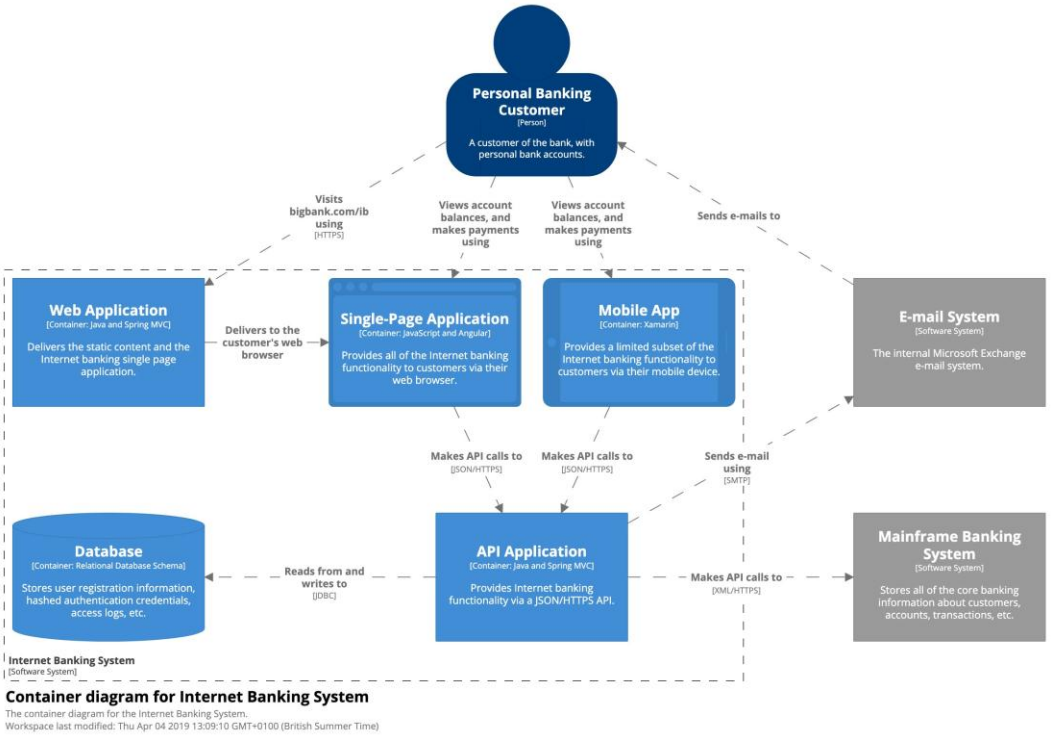


Схема контейнера для системы Интернет-банкинг

Он показывает, что система интернет-банкинга (пунктирное поле) состоит из пяти контейнеров: серверного веб-приложения, одностраничного приложения, мобильного приложения, серверного приложения API и базы данных. Веб-приложение — это веб-приложение Java/Spring MVC, которое обслуживает статический контент (HTML, CSS и JavaScript), включая контент, составляющий одностраничное приложение. Одностраничное приложение — это приложение Angular, которое работает в веб-браузере клиента и предоставляет все функции интернет-банкинга. Кроме того, клиенты могут использовать кроссплатформенное мобильное приложение Xamarin для доступа к подмножеству функций интернет-банкинга.

Как одностраничное приложение, так и мобильное приложение используют JSON/HTTPS API, который предоставляется другим приложением Java/Spring MVC, работающим на сервере. Приложение API получает информацию о пользователе из базы данных (схема реляционной базы данных). Приложение API также

взаимодействует с существующей банковской системой мейнфрейма, используя собственный интерфейс XML/HTTPS, для получения информации о банковских счетах или совершения транзакций. Приложение API также использует существующую Систему электронной почты, если ему необходимо отправлять электронные письма клиентам.

Стоит отметить, что на этой диаграмме ничего не говорится о количестве физических экземпляров каждого контейнера. Например, приложение API может выполняться на ферме серверов Apache Tomcat, а реляционная база данных — в кластере Oracle, но на этой схеме не отображается такой уровень информации. Вместо этого я показываю физические экземпляры, отработку отказа, кластеризацию и т. д. на отдельной диаграмме развертывания, которая иллюстрирует сопоставление контейнеров с инфраструктурой.

Сколько деталей?

Если вы рисуете схему контейнера во время предварительного проектирования, у вас может не быть под рукой некоторых технических деталей. Это нормально, просто добавьте то, что знаете. Если, с другой стороны, вы рисуете диаграмму для документирования существующей системы, более вероятно, что вы сможете добавить некоторые более мелкие детали; такие как протоколы, номера портов и т.д. Выбор за вами, добавьте столько деталей, сколько считаете нужным.

7.3 Элементы

Схема контейнера обычно включает в себя три типа элементов; люди, программные системы и контейнеры.

Контейнеры

Контейнер обычно представляет собой приложение или хранилище данных. Я соберу следующую информацию о каждом контейнере:

- Имя: Имя контейнера (например, "Веб-сервер с выходом в Интернет", "База данных" и т.д.).

- Технология: Технология реализации (например, приложение Java/Spring MVC, веб-приложение ASP.NET, служба Windows C#, схема реляционной базы данных и т.д.).
- Описание: Краткое описание или, возможно, список ключевых обязанностей контейнера или хранящихся сущностей/таблиц/файлов и т.д.

Если вы приступаете к предварительному проектированию и не совсем знаете, какую технологию вы собираетесь использовать для создания конкретного контейнера, то, возможно, вместо этого упомяните короткий список выбранных технологий или общий тип технологии (например, «Схема реляционной базы данных»). С другой стороны, если контейнер представляет собой стандартную базу данных/схему СУБД и не использует какие-либо проприетарные функции, вам не обязательно указывать технологию. Вместо этого вы можете упомянуть технологию на схеме развертывания. Но если ваша база данных использует определенные функции СУБД (например, хранимые процедуры Oracle) и не может быть развернута на другой СУБД без изменений, то вам следует указать технологию, чтобы прояснить это.

Файловые системы и файлы журналов в сравнении с хранением данных

Если я описываю программную систему, в которой приложение хранит критически важные для бизнеса данные в файловой системе, я включаю контейнер «Файловая система» на диаграмму, чтобы сделать это явным. В отличие от этого, хотя многие типы контейнеров записывают файлы журналов в файловую систему (и это, безусловно, важно), я обычно опускаю эту деталь для краткости.

Хранение данных как услуга

Часто задаваемый вопрос заключается в том, следует ли отображать на схеме контейнера такие службы, как Amazon S3, Amazon RDS или Azure SQL Database. В конце концов, это внешние сервисы, которыми мы не владеем и не управляем сами.

Если вы создаете программную систему, которая использует Amazon S3 для хранения данных, это правда, что вы не используете S3 самостоятельно, но у вас есть право собственности и ответственность за используемые корзины. Аналогично с Amazon RDS у вас есть (более или менее) полный контроль над любыми схемами баз данных, которые вы создаете. По этой причине я обычно показываю такие контейнеры на

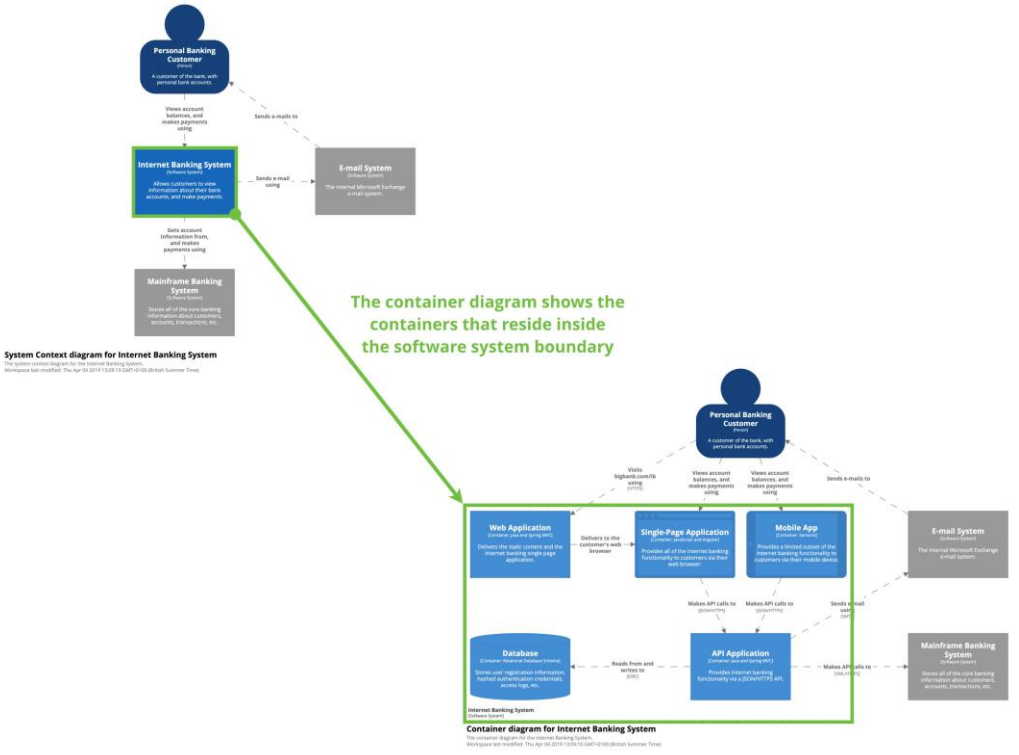
схеме контейнера. Они являются неотъемлемой частью архитектуры вашего программного обеспечения, хотя и размещены в другом месте.

Люди и программные системы

Если вы вернетесь к примеру диаграммы контейнера, то заметите, что в ней задействованы те же люди и программные системы, что и в примере диаграммы системного контекста. Обычно я включаю одну и ту же коллекцию людей (пользователей, актеров, персонажей и т.д.) и программные системы, потому что это помогает рассказать общую картину того, как работает система, и обеспечивает преемственность между двумя диаграммами. По сути, диаграмма контейнера очень похожа на диаграмму системного контекста.
где вы увеличиваете масштаб, чтобы увидеть, что находится внутри программной системы в области действия.

Границы программной системы

Помня об этом, я рекомендую нарисовать ограничивающий прямоугольник вокруг контейнеров на вашей схеме, чтобы явно показать границу системы. Эта граница системы должна соответствовать одному прямоугольнику, отображаемому на схеме системного контекста.



На диаграмме контейнера показано увеличение границы программной системы

7.4 Взаимодействия

Как правило, взаимодействие между контейнерами является внепроцессным (или межпроцессным). Очень полезно четко обозначить это и обобщить, как будут работать эти взаимодействия. Как и в случае с любой диаграммой, я рекомендую аннотировать все взаимодействия, а не иметь диаграмму с набором прямоугольников и неоднозначных линий без надписей, соединяющих все вместе. Полезная информация для аннотирования взаимодействий включает в себя:

- Цель взаимодействия (например, «читает/записывает данные из», «отправляет отчеты» и т. д.).

- Механизм взаимодействия (например, веб-службы, REST, Web API, удаленный вызов методов Java, Windows Communication Foundation, Java Message Service).
- Стиль коммуникации (например, синхронный, асинхронный, пакетный, двухфазный коммит и т. д.).
- Протоколы и номера портов (например, HTTP, HTTPS, SOAP/HTTP, SMTP, FTP, RMI/IIOP и т. д.).

7.5 Мотивация

В то время как диаграмма системного контекста показывает вашу программную систему в виде одного блока, диаграмма контейнера открывает это окно, чтобы показать, что находится внутри него. Это полезно, потому что:

- Это делает выбор технологий высокого уровня явным.
- Он показывает отношения между контейнерами и то, как эти контейнеры взаимодействуют.

Во время моих семинаров по созданию эскизов архитектуры программного обеспечения я часто вижу группы, создающие высокоуровневую контекстную диаграмму, которая показывает рассматриваемую программную систему в виде одного черного ящика, и вторую диаграмму, которая показывает все компоненты, находящиеся внутри всей программной системы. Эта вторая диаграмма обычно запутана и запутанна, потому что на ней представлено слишком много информации. Кроме того, обычно неясно, как компоненты группируются друг с другом с точки зрения развертывания. Диаграмма контейнера представляет собой красивую промежуточную диаграмму между двумя контрастными уровнями абстракции. Это также логически приводит вас к уровню 3 (диаграмма компонентов), где вы открываете один контейнер для отображения только компонентов, которые находятся в этом контейнере.

7.6 Аудитория

Технические специалисты внутри и за пределами непосредственной команды разработчиков программного обеспечения; включая всех, от разработчиков программного обеспечения до операционного персонала и службы поддержки.

7.7 Обязательно или необязательно?

Все программные системы должны иметь диаграмму контейнера.

8. Уровень 3: Схема компонентов

После диаграммы контейнера, показывающей высокоуровневые технологические элементы, следующим шагом является увеличение масштаба и дальнейшая декомпозиция каждого контейнера, чтобы показать компоненты внутри него.

8.1 Намерение

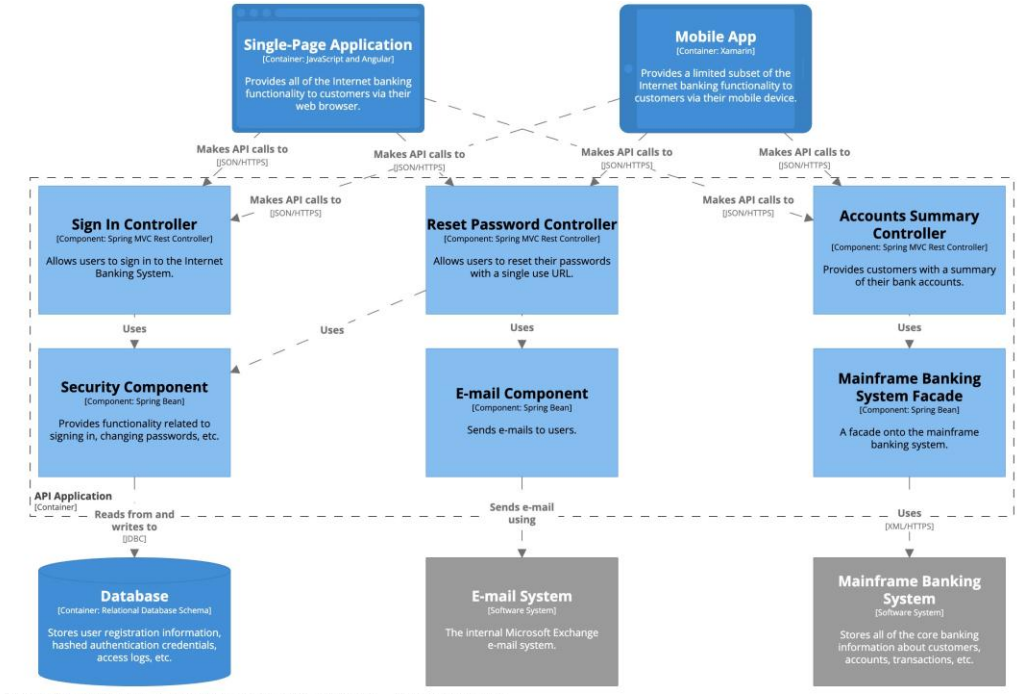
Диаграмма компонентов помогает ответить на следующие вопросы.

А. Из каких компонентов состоит каждый контейнер?
Б. У всех ли компонентов есть дом (т.е. они находятся в контейнере)?
В. Понятно, как работает софт на высоком уровне?

8.2 Структура

Всякий раз, когда людей просят нарисовать «архитектурные схемы», они обычно заканчивают рисованием диаграмм, показывающих компоненты, составляющие их программную систему. В основном это то, что показывает диаграмма компонентов, за исключением того, что мы хотим видеть только компоненты, которые находятся в одном контейнере за раз.

Это пример диаграммы компонентов для вымышленной системы интернет-банкинга, показывающая некоторые (а не все) компоненты в приложении API.



Component diagram for Internet Banking System - API Application

The component diagram for the API Application.

Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

Здесь есть три контроллера Spring MVC Rest, обеспечивающие точки доступа для JSON/HTTPS
API, при этом каждый контроллер впоследствии использует другие компоненты для доступа к данным из базы данных и банковской системы основного фрейма или отправки электронных писем. В целом, на этой схеме показано, как приложение API разделено на компоненты, что из себя представляет каждый из этих компонентов, их обязанности и детали технологии/реализации.

Сколько деталей?

Если вы рисуете схему компонентов во время предварительного проектирования, у вас может не быть под рукой некоторых технических деталей. Еще раз, не переживайте, добавляйте то, что знаете. Если, с другой стороны, вы рисуете диаграмму для документирования существующей системы, у вас под рукой будут более мелкие детали; Например, платформы, которые вы используете для реализации

компонента. Как и в случае со многими другими аспектами диаграмм, выбор степени детализации остается за вами.

8.3 Элементы

Компонентная схема может включать в себя четыре типа элементов; люди, программные системы, контейнеры и компоненты.

Компоненты

Как мы видели при создании нашего общего словаря, компоненты — это крупнозернистые строительные блоки вашей программной системы, которые находятся внутри контейнера. Я соберу следующую информацию для каждого компонента:

- Name: Имя компонента.
- Технология: Технология реализации компонента (например, Plain Old [Java|Язык C#|Ruby|etc] Объект, Enterprise JavaBean, сервис Windows Communication Foundation и т.д.).
- Описание: Краткое описание компонента, обычно краткое предложение с описанием обязанностей компонента.

Вы можете продумать и идентифицировать компоненты независимо от того, как упакован код и какой архитектурный стиль используется. Учитывая это, диаграмма компонентов должна отражать используемый архитектурный стиль; будь то многоуровневая архитектура, шестиугольная архитектура или что-то совершенно другое.

Компоненты инфраструктуры и межсекторальные проблемы

Компоненты инфраструктуры являются важными частями большинства программных систем, но вы можете захотеть включить или не захотеть включать их в схему компонентов. Например, если у вас есть компонент ведения журнала, он, скорее всего, будет использоваться большинством других компонентов в контейнере. Рисование этого компонента и всех взаимодействий может привести к созданию очень запутанной диаграммы. У вас есть несколько вариантов решения этой проблемы:

А. Не включайте компонент ведения журнала, если он не приносит большой пользы в рассказе истории.
Б. Напишите примечание на схеме, которое гласит что-то вроде: «Все компоненты, показанные здесь, записывают логи через компонент ведения журнала».
В. Включите компонент ведения журнала в схему, но не показывайте, как он связан с чем-либо еще. Вместо этого аннотируйте элементы, использующие компонент логирования (с помощью символа/значка), или используйте цветовую кодировку, которую вы затем можете описать в ключе/легенде диаграммы (например, «звездочка обозначает связь с компонентом логирования»).

Общие компоненты и библиотеки

Меня часто спрашивают, должна ли диаграмма компонентов включать в себя общие компоненты (например, из общей или статической библиотеки) и как такие компоненты должны быть представлены. Если включение общего компонента помогает рассказать историю, то он обязательно должен быть включен. Если вы хотите проиллюстрировать, что определенный компонент является общим компонентом или находится в определенной библиотеке/модуле, опять же, вы можете использовать обозначение (например, символ/иконку или цветовую кодировку) для представления этого факта.

Многокомпонентные реализации

Во многих случаях существует только одна реализация компонента. Тем не менее, могут быть случаи, когда у вас будет интерфейс с одним компонентом и несколько реализаций. Это особенно верно в отношении программных продуктов (а не заказного программного обеспечения),
где набор активных компонентов будет выбран через конфигурацию при установке продукта. В качестве примера можно привести различные реализации компонентов хранения данных (например, одну для Microsoft SQL Server, одну для MySQL и т. д.), различные компоненты ведения журнала (например, локальный диск или очередь сообщений) или подключаемые компоненты аутентификации для интеграции с различными поставщиками удостоверений.

Наличие многокомпонентных реализаций поднимает вопрос о том, как это должно быть проиллюстрировано на диаграмме компонентов. Если мы возьмем пример компонента ведения журнала с несколькими реализациями (например, локальный диск или очередь сообщений), одна из которых выбирается во время развертывания с помощью конфигурации, есть несколько подходов к этому представлению:

A. Первый подход заключается в том, чтобы опустить факт существования нескольких реализований компонентов и нарисовать диаграмму компонентов, как если бы существовала только одна реализация. Здесь я бы нарисовал компонент логирования и описал его обязанности, которые в данном случае могут заключаться в «логировании ошибок и других системных событий».
Б. Если бы я хотел включить тот факт, что реализация компонента может быть различной, я мог бы добавить дополнительный текст в поле компонента протоколирования на диаграмме, чтобы сказать что-то вроде: "Записи журнала хранятся на локальном диске или отправляются в очередь сообщений, в зависимости от реализации, выбранной конфигурацией во время развертывания". Вы также можете добиться того же результата, выделив компонент ведения журнала с помощью символа/значка или цветового кодирования. Затем ключ к диаграмме объяснил бы это.
B. Другой подход заключается в том, чтобы иметь одну диаграмму компонентов для каждого варианта реализации. Это работает лучше всего, если у вас есть только небольшое количество комбинаций реализации компонентов (например, у вас есть только один или два компонента, в которые реализация может быть заменена во время выполнения). Наличие отдельных диаграмм для реализаций конкретных компонентов также может быть полезно, если эти реализации компонентов сами вводят другие компоненты, которые в противном случае не были бы видны на диаграмме.

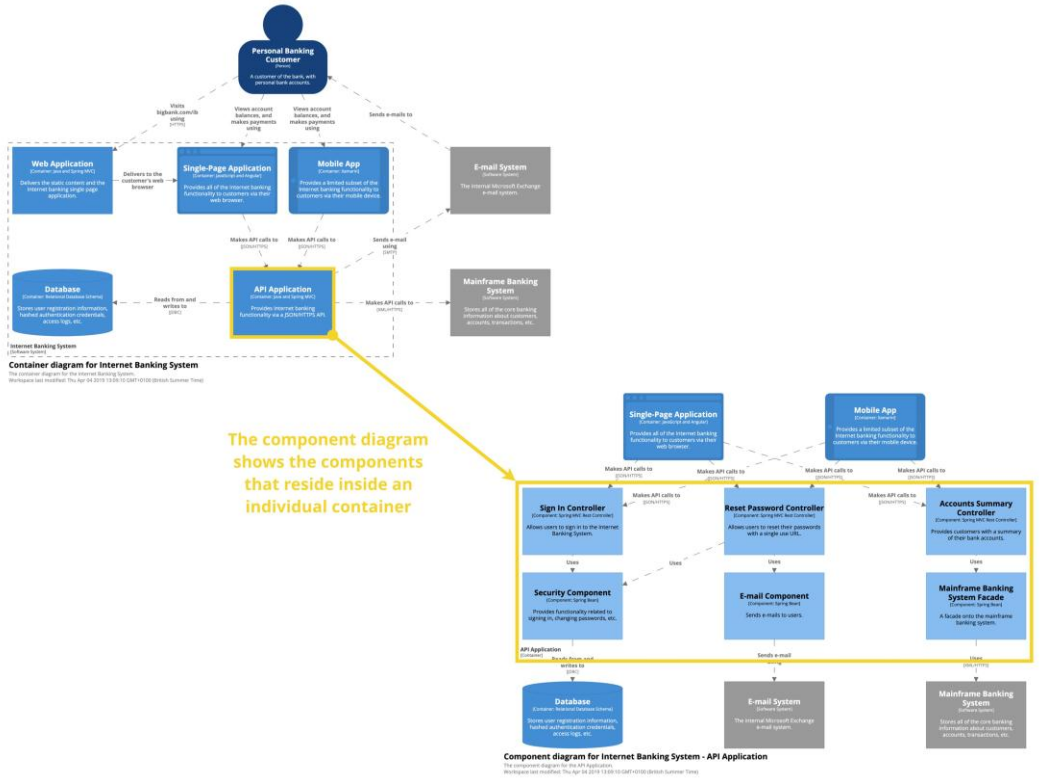
Как и во многих других вещах, обсуждаемых в этой книге, нет «правильного» ответа, и это действительно зависит от того, какую историю вам нужно рассказать.

Люди, программные системы и контейнеры

Как и в случае со схемой контейнера, может быть полезно включить людей, программные системы и другие контейнеры, чтобы помочь создать некоторый контекст вокруг контейнера, который вы увеличили. Компонентная схема системы интернет-банкинга для приложения API включает в себя другие контейнеры и программные системы, с которыми оно взаимодействует.

Граница контейнера

Если вы собираетесь включить людей, программные системы и контейнеры в схему компонентов
(и, опять же, вы должны это сделать), я рекомендую нарисовать ограничивающий прямоугольник, чтобы явно показать границу контейнера.



На схеме компонентов показано увеличение масштаба одного контейнера

8.4 Взаимодействия

Повторим тот же совет, который дается для других типов диаграмм, полезно аннотировать взаимодействия между компонентами, а не иметь диаграмму с набором прямоугольников и неоднозначных линий, соединяющих их все вместе. Полезная информация для добавления диаграммы включает в себя:

- Цель взаимодействия (например, «использует», «сохраняет данные с использованием», «делегирует» и т. д.).
- Стиль общения (например, синхронный, асинхронный и т.д.).

8.5 Мотивация

На схеме компонентов отображаются компоненты, которые находятся внутри отдельного контейнера. Это полезно, потому что:

- Он показывает высокоуровневое разложение контейнера на компоненты, каждый из которых имеет свои обязанности.
- Он показывает, где существуют отношения и зависимости между компонентами.
- Он предоставляет высокоуровневую сводку деталей реализации, включая любые используемые фреймворки или библиотеки.
- Если компоненты, показанные на схеме, могут быть явно сопоставлены с кодом, у вас есть хороший способ по-настоящему понять структуру кодовой базы.

8.6 Аудитория

Технические специалисты в команде разработчиков программного обеспечения.

8.7 Обязательно или необязательно?

Обычно я не рисую диаграмму компонентов для контейнеров хранения данных (например, баз данных, файловых систем, хранилищ контента и т. д.) или для тех контейнеров, которые очень просты по своей природе (например, микросервисы). Для монолитных приложений можно рассмотреть возможность создания схемы компонентов, особенно в процессе предварительного проектирования. В реальном мире многие группы считают этот уровень детализации необязательным для долговременной документации из-за потенциально высокой степени усилий, затрачиваемых на создание схем и поддержание их в актуальном состоянии. Кроме того, как только у вас будет больше нескольких компонентов, вы обнаружите, что диаграмма очень быстро начинает загромождаться, что снижает ее полезность. Мы рассмотрим некоторые стратегии решения этой проблемы позже в этой книге.

9. Уровень 4: Диаграммы на уровне кода

Последний уровень детализации — это одна или несколько диаграмм на уровне кода, показывающих детали реализации отдельного компонента.

9.1 Намерение

Цель диаграммы на уровне кода — проиллюстрировать структуру кода и, в данном случае, то, как реализован компонент.

9.2 Структура

Если вы используете объектно-ориентированный язык программирования, вероятно, лучший способ сделать это — использовать диаграмму классов UML, либо автоматически генерируя ее из кода, либо рисуя от руки. Рассмотрим пример, увеличив компонент "Mainframe Banking System Facade" из контейнера "API Application" системы интернет-банкинга.

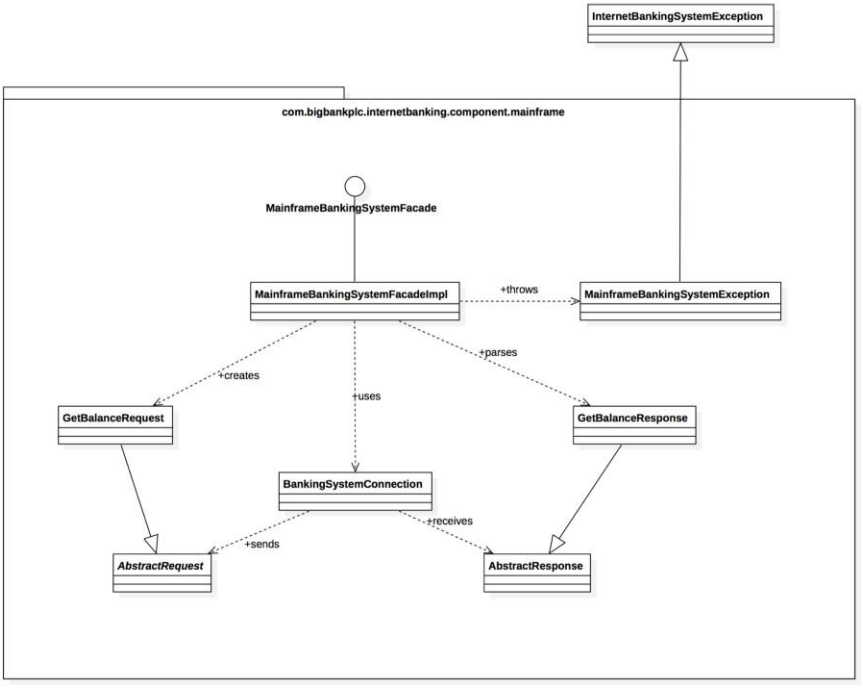


Диаграмма классов UML

Сколько деталей?

Опасность диаграмм классов UML заключается в том, что в них очень легко включить значительное количество деталей, и это особенно верно, если вы автоматически генерируете диаграммы из кода. Несмотря на то, что заманчиво включить все поля/свойства/атрибуты и методы, я бы не поддался этому искушению и включил бы только столько информации, сколько вам нужно, чтобы рассказать историю, которую вы хотите рассказать. Как правило, я включаю только те атрибуты и методы, которые имеют отношение к повествованию, которое я хочу создать.

UML-диаграммы классов часто используются для описания всего приложения, но это приводит к огромному беспорядку из перекрывающихся блоков и линий, независимо от того, насколько хорошо структурирован код. Ключом к использованию диаграмм классов является ограничение их области применения. В этом случае область применения ограничена внутренними компонентами.

Уровень 4: Диаграммы на уровне кода

9.3 Мотивация

Наличие этого последнего уровня абстракции позволяет сопоставлять высокоуровневые, крупнозернистые компоненты с реальными элементами кода. Это помогает соединить то, что иногда воспринимается как два совершенно разных мира; архитектура программного обеспечения и код.

9.4 Аудитория

Технические специалисты в команде разработчиков программного обеспечения, в частности, разработчики программного обеспечения.

9.5 Обязательно или необязательно?

Поскольку этот уровень детализации заложен в коде, разработчики программного обеспечения могут получить эти сведения по запросу из самого кода. Таким образом, это определенно необязательный уровень детализации, и в целом не рекомендуется. Обычно я не рисую диаграммы классов для чего-либо, кроме самых сложных компонентов, или в качестве шаблона, когда хочу описать шаблон, используемый в кодовой базе.

10. Нотация

Теперь, когда мы создали общий словарь и рассмотрели, как создавать диаграммы на нескольких различных уровнях абстракции, давайте посмотрим на нотацию.

10.1 Титулы

Первое, что действительно может помочь людям понять диаграмму, — это включение заголовка. Если вы используете нотацию, такую как UML, элементы диаграммы, вероятно, дадут ключ к пониманию контекста диаграммы. Это не очень помогает, если у вас есть коллекция диаграмм, которые представляют собой просто квадраты и линии.

Включайте короткий и содержательный заголовок на каждую диаграмму, даже если вы используете UML. И если диаграммы должны читаться в определенном порядке,

убедитесь, что это ясно указано в заголовке, возможно, с помощью схемы нумерации. Чтобы избежать путаницы, я также рекомендую включить тип диаграммы в заголовок. Например, что-то вроде:

- Контекстная диаграмма системы финансовых рисков
- [Системный контекст] Система финансовых рисков

10.2 Ключи и легенды

Одним из преимуществ использования такой нотации, как UML, является то, что она предоставляет стандартизированный набор элементов диаграммы для каждого типа диаграммы. Теоретически, если кто-то знаком с этими элементами, он должен быть в состоянии понять вашу диаграмму. В реальном мире это не всегда так, но это определенно не относится к диаграммам «квадраты и линии», где люди, рисующие диаграммы, изобретают обозначения по ходу дела.

Нет ничего плохого в том, чтобы придумать свою собственную нотацию, но убедитесь, что вы даете всем равные шансы понять ее, включив ключ/легенду где-нибудь на диаграмме или рядом с ней. Вот некоторые вещи, которые вы, возможно, захотите включить в пояснения:

- Формы

- Стили линий
- Цвета
- Границы
- Сокращения

Часто можно делать предположения и интерпретировать использование элементов диаграммы без ключа. Например, я слышал, как люди говорят что-то подобное во время моих семинаров по созданию эскизов архитектуры программного обеспечения:

> «Серые ящики кажутся существующими системами, а красные ящики — новыми системами»

Даже если обозначение кажется вам очевидным, я рекомендую перестраховаться и добавить ключ/легенду. Даже, казалось бы, очевидное может быть неверно истолковано людьми с разным образованием и опытом.

10.3 Элементы

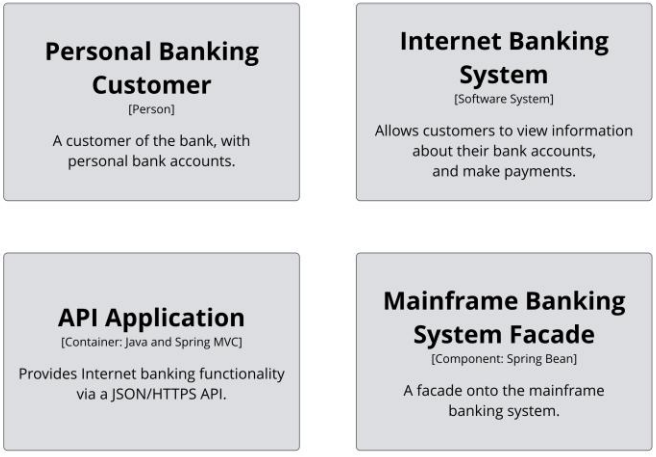
Большинство диаграмм «коробок и линий», которые я видел, — это не просто коробки и линии, в которых команды используют различные формы для представления элементов в архитектуре своего программного обеспечения. Например, вы часто будете видеть цилиндры на диаграмме, и многие люди будут интерпретировать их как базу данных с каким-либо описанием. Но это не всегда так.

Я рекомендую начать с чистой диаграммы «квадраты и линии», используя очень утилитарную нотацию, а затем добавить формы, цвет и границы, чтобы добавить дополнительную информацию или сделать диаграмму более эстетически привлекательной. Для того, чтобы показать некоторые примеры диаграмм архитектуры программного обеспечения в этой книге, мне нужно было создать свою собственную нотацию, которая включает в себя следующую информацию для каждого элемента:

- Персона: Имя и описание.
- Система программного обеспечения: Название и описание.
- Контейнер: Название, технология и описание.

- Компонент: Название, технология и описание.

Как вы могли видеть из примеров диаграмм, каждый из элементов рисуется следующим образом:



Примеры элементов, которые я использую в своих диаграммах

Это обозначение, на котором я постепенно остановился на протяжении многих лет. Его легко рисовать на доске или в инструменте, а также он хорошо работает на стикерах и карточках. Не стесняйтесь
Тем не менее, создайте свою собственную нотацию.

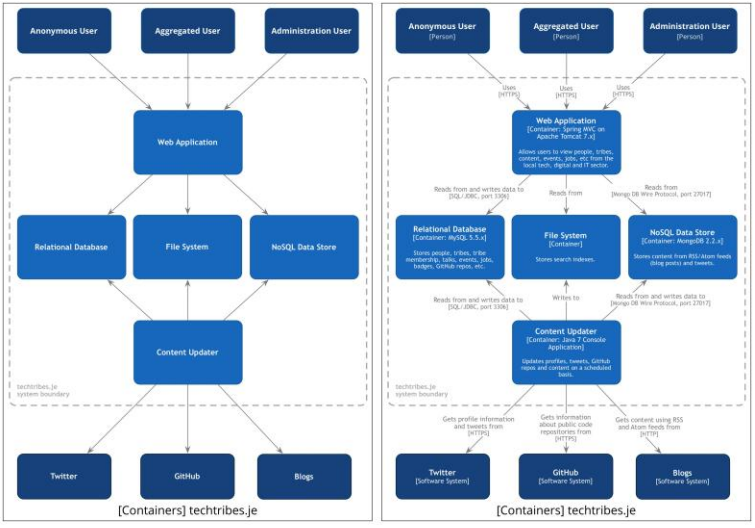
Описание/обязанности

Если именование является одной из самых сложных вещей в разработке программного обеспечения, не поддавайтесь искушению иметь диаграмму, полную коробок, содержащих только имена. Если вы посмотрите на большинство диаграмм архитектуры программного обеспечения, то увидите, что это именно то, чем они являются - набор именованных блоков. Как и во многих других вещах, именование всегда открыто для интерпретаций и двусмысленности.

Действительно простой, но эффективный способ добавить дополнительный слой информации к архитектурной схеме и устранить двусмысленность из нее — это аннотировать элементы диаграммы с кратким описанием того, в чем заключаются их

обязанности. Маркированный список (7 +/- 2 товаров) или короткое предложение хорошо работает.

При условии, что он будет кратким (и использование более мелкого шрифта для этой информации также может помочь), добавление большего количества текста на диаграммы может помочь получить действительно полезное представление о том, что делает программная система и как она была структурирована. Взгляните на следующие диаграммы, какие из них вы предпочитаете?



Добавление дополнительного описательного текста к элементам схемы может устранить двусмысленность

Цвет

Схемы архитектуры программного обеспечения не обязательно должны быть черными, белыми и различными оттенками серого. Использование цвета — отличный способ дополнить схему, которая уже имеет смысл. Например, цвет может быть использован для обеспечения дифференциации между элементами диаграммы или для того, чтобы на них не делался акцент, и вы можете кодировать элементы цветом в соответствии с:

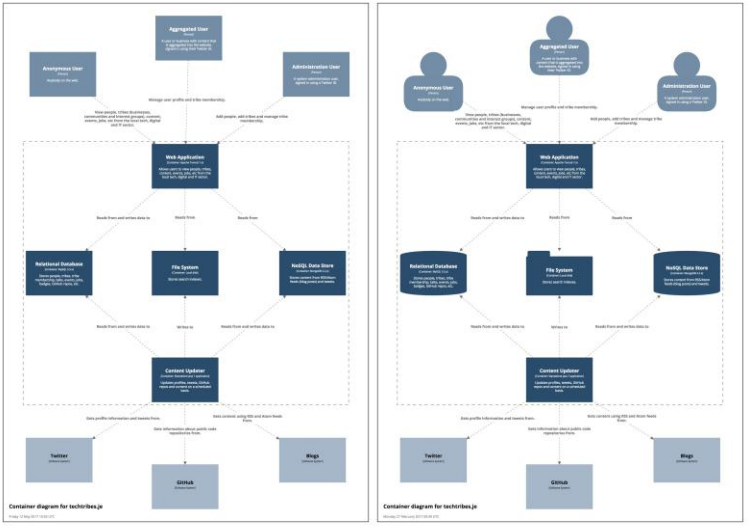
- Существующие и новые.

- Готовый продукт против индивидуальной сборки.
- Тип технологии или платформы.
- Профиль риска (например, риск строительства; высокий-средний-низкий риск; красный-янтарный-зеленый).
- Размер и/или сложность.
- Право собственности (т.е. элементы, которыми владеете вы, и элементы, принадлежащие кому-то другому).
- Внутренние и внешние (т.е. элементы внутри вашей организации и за ее пределами).
- Элементы, которые вы изменяете или удаляете в следующем релизе/спринте/фазе, по сравнению с теми, которые останутся нетронутыми.

Если вы собираетесь использовать цвет, а я рекомендую вам это сделать, убедитесь, что очевидно, какова ваша схема цветового кодирования, включив ссылку на эти цвета в ключ/легенду.
Цвет может изменить мир к лучшему. Просто помните о том, что в вашей команде есть люди, страдающие дальтонизмом, и убедитесь, что ваша цветовая схема работает, если диаграмма будет печататься на черно-белых принтерах.

Формы

Использование различных форм может быть отличным способом добавить дополнительный уровень информации, дополнить, улучшить или подчеркнуть определенные элементы. Использование фигур также может сделать диаграмму более эстетичной. Хотя это звучит поверхностно, люди с большей вероятностью будут смотреть на диаграммы, которые приятны для глаз. Рассмотрим две версии схемы контейнера techtribes.je, которые приведены ниже. Они оба показывают одну и ту же информацию для каждого элемента (имя, тип элемента, описание и технология, если применимо), но в одном используются только рамки, в то время как в другом используются некоторые формы.



Диаграмму, в которой используются фигуры, безусловно, легче читать на расстоянии, а формы помогают обеспечить более быстрый просмотр «с первого взгляда». Но формы просто улучшают диаграмму; На самом деле они не добавляют никакой информации, которая уже присутствует в тексте, который находится внутри элементов.

Унифицированный язык моделирования имеет множество типов диаграмм и еще большее количество типов элементов, которые могут отображаться на этих диаграммах. Как ни странно, интерпретация нотации является одной из основных причин, приводимых для отказа от использования UML, и многие разработчики программного обеспечения говорили мне, что в нотации просто слишком много типов диаграмм и нюансов.

В отличие от этого, я предпочитаю использовать очень простую нотацию, состоящую из небольшого количества форм. За многие годы, что я веду свой семинар по эскизам архитектуры программного обеспечения, я заметил, что большинство разработчиков обычно используют на своих диаграммах только следующие формы:

- Коробки (квадраты, прямоугольники, закругленные коробки, круги, эллипсы и шестиугольники).

- Фигуры людей («человечек-палка»,[1] «голова и плечи» и т.д.).
- Цилиндры (например, для представления баз данных).
- Формы папок (например, для представления файловых систем).

Мой совет - делайте диаграммы как можно проще, но не стесняйтесь использовать любые формы, которые вам нравятся. Опять же, не забудьте включить фигуры на ключ/легенду.

Границы

Как и в случае с фигурами, добавление границ (например, двойных линий, цветных линий, пунктирных линий и т. д.) вокруг элементов диаграммы может быть отличным способом сделать акцент или сгруппировать связанные элементы вместе. Если вы это сделаете, убедитесь, что вам понятно, что означает граница, либо обозначив границу, либо добавив пояснение к ключу/легенде.

Размер

Небольшое замечание о размере элементов. Будьте осторожны при выборе размеров элементов на диаграммах. Я был свидетелем того, как люди склонны делать предположения об элементах, размер которых отличается от других. Более крупные элементы часто считаются более крупными, более сложными или более значительными; в то время как более мелкие элементы имеют тенденцию приобретать обратные характеристики. Если вы специально не делаете заявление о размере, сложности или значимости, я рекомендую рисовать все элементы примерно одного размера.

10,4 Строки

Линии являются важной частью большинства архитектурных схем, выступая в качестве клея, который удерживает все коробки вместе. Большая проблема с линиями заключается именно в этом - они, как правило, воспринимаются как неважные вещи,

[1] Возможно, вы видели диаграммы, на которых программные системы представлены в качестве акторов с использованием традиционного значка «человечка-палки». Это происходит из UML, где «актор указывает роль, которую играет пользователь или любая другая система, которая взаимодействует с субъектом». Я сам делал это в прошлом, но сейчас стесняюсь этого делать, так как это может вызвать слишком много путаницы. В конце концов, зачем вам визуально представлять программную систему с помощью формы человека?

которые удерживают другие, более значимые элементы диаграммы вместе. В результате линиям часто не уделяется особого внимания.

Направленность

Несмотря на то, что большинство отношений между элементами являются двунаправленными (например, запрос, за которым следует ответ, или поток данных в обоих направлениях), я обычно выбираю наиболее значимое направление и представляю его в виде однонаправленной линии. В связи с этим возникает вопрос: «В какую сторону вы указываете стрелки?».

В techtribes.je примере на диаграмме контекста мои пользователи используют techtribes.je, который в свою очередь использует ряд других программных систем. Если мы посмотрим на границу между techtribes.je и Twitter, я мог бы провести ее несколькими способами, в зависимости от того, хочу ли я показать отношения зависимости или поток данных:

- [techtribes.je] – получает информацию профиля и твиты от -> [Twitter]
- [techtribes.je] – получает информацию профиля и твиты от -> [Twitter]
- [Twitter] – отправляет информацию профиля и твиты на -> [techtribes.je]

В этом примере нет никакого «правильного ответа». Лично я предпочитаю показывать отношения зависимости и проводить линию от инициатора к получателю. Другие варианты также действительны, и стиль, который вы выберете, остается вашим решением. Мой совет — будьте последовательны там, где это возможно, и убедитесь, что описательный текст, который вы используете для аннотации строки, соответствует тому, что вы пытаетесь описать.

Описание

Что касается формулировок описаний в строках, я постараюсь использовать формулировки, которые помогают объяснить направление стрелки, часто используя или заканчивая описание предлогом. Предлог — это тип слова, который выражает что-то об отношениях между элементами предложения (например, к, от, с, на, в). Например, напишу:

- [techtribes.je] – получает информацию профиля и твиты от -> [Twitter]
- [Веб-приложение] – считывает и записывает в -> [База данных]

А не:

- [techtribes.je] – получает информацию профиля и твиты -> [Twitter]
- [Веб-приложение] – чтение и запись -> [База данных]

Чтобы проверить, имеет ли смысл описание отношений, просто произнесите полученное предложение вслух. Например, «techtribes.je получает информацию профиля и твиты из Twitter». Если предложение не имеет смысла, вам, вероятно, нужно изменить формулировку.

Тип линии

Как и в случае с элементами, вы можете использовать различные стили и цвета линий, чтобы добавить дополнительный уровень информации в свою диаграмму. Например, возможно, синхронные взаимодействия иллюстрируются сплошными линиями, в то время как асинхронные взаимодействия иллюстрируются пунктирными линиями. И, возможно, HTTPS-соединения окрашены в зеленый цвет, а HTTP-соединения — в желтый.

Еще раз убедитесь, что любой стиль дополняет существующую информацию, где это возможно, и что используемые вами стили описаны на ключе/легенде.

Одна линия против многих

Обычно между элементами диаграммы существует несколько взаимосвязей. В techtribes.je примере techtribes.je получает как информацию профиля, так и твиты из Twitter. Эта связь отображается в виде одной линии, но на самом деле здесь используются два API. Если вы хотите быть более точным, не стесняйтесь использовать две отдельные линии, чтобы проиллюстрировать две разные взаимосвязи:

- [techtribes.je] – получает информацию профиля из -> [Twitter]
- [techtribes.je] – получает твиты от -> [Twitter]

10.5 Макет

Использование электронных инструментов рисования упрощает позиционирование элементов диаграммы, поскольку вы можете перемещать их столько, сколько захотите. Многие люди предпочитают разрабатывать программное обеспечение, стоя

перед доской или флипчартом, особенно потому, что это обеспечивает более широкую и лучшую среду для совместной работы. Компромисс здесь заключается в том, что вам придется больше думать о расположении элементов диаграммы, потому что это может стать неудобным, если вам придется постоянно рисовать, стирать и перерисовывать элементы ваших диаграмм, когда у вас заканчивается место.

Стикеры и карточки могут помочь вам обеспечить некоторую гибкость, если вы используете их в качестве замены рамок для рисования. А если вы используете технику в стиле «Класс-Ответственность-Сотрудничество » для определения классов/компонентов/сервисов-кандидатов во время сеанса проектирования, вы можете использовать полученные индексные карточки в качестве способа начать создание своих диаграмм.

Примеры использования стикеров и карточек вместо рамок для рисования

Нужно переместить некоторые элементы? Нет проблем, просто переместите их. Нужно удалить некоторые элементы? Нет проблем, просто уберите их со схемы и выбросьте. Стикеры и карточки могут быть отличным способом начать работу со схемами архитектуры программного обеспечения, но я считаю, что полученные диаграммы могут выглядеть загроможденными. Да, и стикеры часто плохо прилипают
доски, так что имейте под рукой немного Blu-Tack !

10.6 Ориентация

Представьте, что вы разрабатываете 3-уровневое веб-приложение, состоящее из веб-уровня, промежуточного уровня и базы данных. Если вы рисуете схему контейнера, то с какой стороны вы ее рисуете? Пользователи и веб-уровень вверху с базой данных внизу? Наоборот? Или, может быть, вы выкладываете элементы слева направо?

На большинстве архитектурных диаграмм, которые я вижу, пользователи и веб-уровень находятся наверху, но это не всегда так. Иногда те же самые диаграммы будут представлены вверх ногами или задом наперед, возможно, иллюстрируя мнение автора (потенциально подсознательное) о том, что база данных является центром их вселенной. Несмотря на то, что не существует «правильной» ориентации, рисование диаграмм «вверх ногами» по сравнению с тем, что мы могли бы считать нормой, может либо сбивать с толку, либо использоваться с большим эффектом. Выбор за вами.

Я рекомендую вам попытаться поместить самое важное в центр вашей диаграммы и обойти это. Кроме того, постарайтесь, чтобы расположение элементов было одинаковым между диаграммами. Например, все люди на моих techtribes.je диаграммах расположены вверху, а мои системные зависимости — внизу.

10.7 Аббревиатуры

Скорее всего, на ваших диаграммах будет несколько меток; включая названия программных систем, предметные понятия и терминологию и т.д. По возможности избегайте использования аббревиатур, а если вам все же нужно использовать аббревиатуры для краткости, убедитесь, что они задокументированы в глоссарии или на ключе/легенде. В то время как обычные члены команды могут хорошо разбираться в распространенных аббревиатурах доменов, люди, не являющиеся членами команды, или новички, скорее всего, не знают.

Исключением здесь являются аббревиатуры, используемые для описания выбора технологий, особенно если они широко используются в отрасли. В качестве примера можно привести такие вещи, как JMS (Java Message Service), POJO (старый добрый Java-объект) и WCF (Windows Communication Foundation). Пусть ваш конкретный контекст подскажет, нужно ли вам объяснять эти аббревиатуры, и если вы сомневаетесь, перестрахуйтесь и используйте полное название/термин или добавьте аббревиатуру к ключу/легенде.

10.8 Атрибуты качества

В некоторых случаях добавление информации об атрибутах качества дает дополнительную информацию о том, что делает программная система и как она была спроектирована. Самый простой способ сделать это – добавить текст на схему. Например, описание элементов диаграммы может быть расширено за счет включения примечания о количестве потенциальных пользователей, ожидаемом количестве одновременных пользователей и т.д. Строки между элементами также могут содержать дополнительную информацию о передаваемых объемах данных, целевых задержках сообщений, целевом времени запроса/ответа и т. д. Другие сквозные проблемы, такие как безопасность, труднее отобразить на диаграмме, и обычно их лучше оставить для описания в облегченной дополнительной документации.

Когда дело доходит до иллюстрации решений, которые учитывают атрибуты качества, вы можете добавить текст, указывающий, например, что определенные части архитектуры программного обеспечения реплицируются или кластеризуются для решения проблем с масштабируемостью и/или доступностью. Однако мои схемы контекста, контейнеров и компонентов обычно не показывают этого, потому что я сохраняю эту информацию для отдельной диаграммы развертывания, которая показывает, как технология сопоставляется с инфраструктурой.

Мой простой совет заключается в том, что не стоит пытаться втиснуть все в диаграмму. Упрощенная дополнительная документация иногда является лучшим подходом для создания повествования, объясняющего, как учитываются атрибуты качества.

10.9 Область применения диаграммы

Примеры диаграмм, которые я представил в книге, удобно уместились на одной странице. Конечно, в реальном мире вы, скорее всего, будете иметь более крупные и сложные программные системы. По самой своей природе диаграммы компонентов склонны становиться большими и сложными. Так что же делать, если ваша диаграмма не помещается на одной странице?

Простой, но наивный подход заключается в использовании холста большего размера. Если вы работаете на бумаге формата A4, возьмите немного бумаги формата A3 или склейте два листа бумаги формата A4 вместе. Если вы работаете на бумаге формата A3, попробуйте найти бумагу для флипчарта. Если вы работаете с электронным инструментом рисования, просто увеличьте размер страницы. Проблема с

увеличением размера страницы заключается в том, что это позволяет отображать больше информации, что приводит к еще большему беспорядку. Я видел гигантские диаграммы на стенах организаций, которые я посещал, некоторые из них были напечатаны на специальных плоттерных машинах, которые работают с очень большими размерами бумаги.

Даже при наличии относительно небольшой системы программного обеспечения заманчиво попытаться объединить всю историю на одной диаграмме. Например, если у вас есть веб-приложение, кажется логичным создать диаграмму с одним компонентом, на которой показаны все компоненты, составляющие это веб-приложение. Если ваша программная система действительно не настолько мала, вам, скорее всего, не хватит места на холсте диаграммы или вам будет трудно найти макет, который не был бы загромождён мириадами перекрывающихся линий. Иногда может помочь использование большого холста диаграммы, но большие диаграммы обычно трудно интерпретировать и понять из-за слишком высокой когнитивной нагрузки. И если никто не понимает схему, никто не будет на нее смотреть.

В качестве примера этого в действии давайте рассмотрим, как будет выглядеть диаграмма компонентов для techtribes.je веб-приложения, если вы решите включить все компоненты.

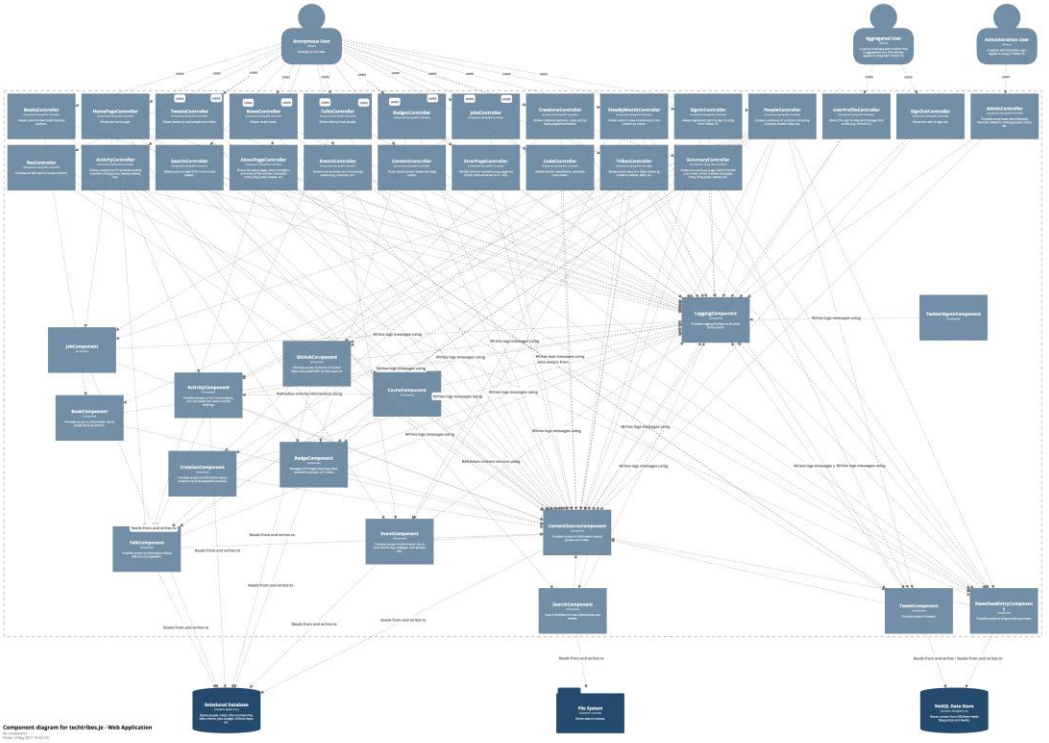


Диаграмма компонентов для веб-приложения techtribes.je

Эта диаграмма компонентов была автоматически создана с помощью инструментов, которые идентифицируют компоненты и их зависимости от кода. Он всеобъемлющий, но это беспорядок! И трудно определить, вызван ли этот беспорядок беспорядком в архитектуре или диаграммой, показывающей слишком много информации. Если посмотреть на эту схему немного внимательнее, то
Мы видим, что на самом деле есть три вещи, которые вызывают загромождение этой диаграммы:

А. Мы показываем каждый контроллер web-MVC (компоненты в верхней части схемы) и, следовательно, каждый путь зависимостей через веб-приложение.
Б. "ContentSourceComponent" используется большим количеством других компонентов (конечно, это может быть хорошо, а может и не быть хорошо с архитектурной точки зрения).
В. "LoggingComponent" используется почти всеми остальными компонентами.

Удаление "LoggingComponent" устранит часть беспорядка, но на самом деле это не очень поможет. Увеличение размера страницы также не поможет. Вместо этого нам нужен другой подход.

Лучшим решением было бы разбить эту сложную диаграмму на большее количество более простых диаграмм, каждая из которых имеет определенную направленность на бизнес-область, функциональную область, функциональную группировку, ограниченный контекст, сценарий использования, взаимодействие с пользователем, набор функций и т. д. Для диаграммы компонентов techtribes.je веб-приложения мы могли бы сделать это, создав одну диаграмму для каждого контроллера web-MVC. Например, вот диаграмма компонентов, которая сосредоточена на "TweetsController" (странице на веб-сайте, которая показывает последние твиты местных жителей и компаний).

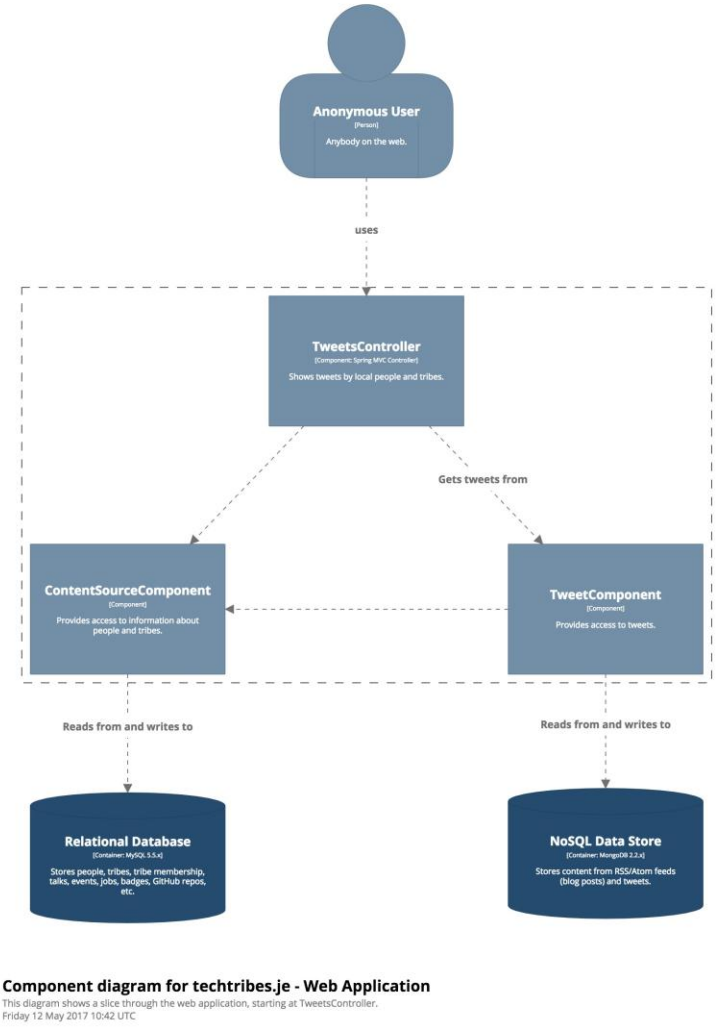


Диаграмма компонентов для веб-приложения techtribes.je, ориентированного на TweetController

Ключ к этому подходу заключается в том, чтобы гарантировать, что каждая из отдельных диаграмм рассказывает разную часть одной и той же общей истории на одном и том же уровне абстракции. Для того, чтобы сделать возможным применение этого подхода, вам действительно нужно использовать некоторые инструменты, но об этом мы поговорим позже.

10.10 Прислушивайтесь к вопросам

В заключение хочу сказать, что будьте внимательны к любым возникающим вопросам или уточнениям, которые могут быть сделаны во время составления или представления диаграмм. Если вы говорите что-то вроде: «Просто для ясности, эти стрелки представляют потоки данных», убедитесь, что эта информация оказывается где-то в ключе/легенде.

11. Диаграммы должны отражать реальность

По-видимому, существует распространенное заблуждение, что «архитектурные диаграммы» должны представлять высокоуровневое концептуальное представление о программной системе, поэтому неудивительно, что разработчики программного обеспечения часто считают их ненужными.

11.1 Разрыв между моделью и кодом

Одна из самых больших проблем, с которыми я вижу диаграммы архитектуры программного обеспечения, заключается в том, что они никогда полностью не соответствуют коду. Иногда диаграммы ужасно устаревают, возможно, потому, что накладные расходы на их обслуживание слишком велики, но в других случаях абстракции, отображаемые на диаграмме, на самом деле не отражают код. И особенно это проявляется на компонентном уровне.

Давайте представим, что вы унаследовали недокументированную кодовую базу, которая представляет собой несколько миллионов строк Java-кода, возможно, разбитого примерно на сто тысяч Java-классов. Предположим, что вам дали задание создать несколько схем архитектуры программного обеспечения, которые помогут описать систему остальным членам команды. С чего начать?

Если у вас достаточно времени и терпения, построение диаграммы классов кодовой базы, безусловно, является вариантом. Хотя к тому времени, когда вы закончите рисовать сто тысяч коробок (по одному на класс), диаграмма, скорее всего, устареет. Автоматизация этого процесса с помощью инструмента статического анализа или построения диаграмм также вряд ли поможет делу. Проблема здесь в том, что информации слишком много для понимания.

Вместо этого мы ищем связанные группы классов и вместо этого рисуем диаграмму, показывающую их. Эти связанные группы классов обычно обозначаются модулями, компонентами, службами, слоями, пакетами, пространствами имен, подсистемами и т. д. То же самое можно сказать, когда вы делаете предварительный дизайн для новой программной системы. Хотя вы могли бы начать с набросков диаграмм классов, это, вероятно, слишком быстрое погружение в детали.

Есть ряд преимуществ в том, чтобы думать о программной системе в терминах больших абстракций, но, по сути, это позволяет нам думать и говорить о программном обеспечении как о небольшом количестве больших вещей, а не о сотнях и тысячах мелочей. Абстракции помогают нам рассуждать о большой и/или сложной программной системе.
Несмотря на то, что при описании программной системы мы можем ссылаться на такие вещи, как компоненты, и многие из нас считают, что наши приложения построены на основе ряда сотрудничающих компонентов, эта структура обычно не проявляется в коде. Это одна из причин, почему существует разрыв между архитектурой программного обеспечения и кодированием как дисциплинами - архитектурные диаграммы на стене говорят одно, а код - другое.

Когда вы открываете кодовую базу, она часто отражает какую-то другую структуру из-за организации кода. Сопоставление между архитектурным представлением программной системы и кодом часто сильно отличается. Вот почему иногда можно увидеть, как люди игнорируют архитектурные диаграммы и говорят: «Код — это единственная точка истины». Джордж Фэрбенкс называет это «разрывом между моделью и кодом» в своей книге под названием «Архитектура программного обеспечения Just Enough».

Предпосылка заключается в том, что, хотя мы думаем о наших программных системах как о состоящих из компонентов, модулей, сервисов, слоев и т. д., у нас нет тех же концепций в языках программирования, которые мы используем. Например, есть ли в Java ключевое слово "component" или "layer"? Нет, наши Java-системы построены из набора классов и интерфейсов, обычно организованных в несколько пакетов. Именно это несоответствие между архитектурными концепциями и кодом может помешать нашему пониманию.

Это не новая проблема. Если вы попросите разработчика программного обеспечения нарисовать диаграмму для описания программной системы, над которой он работает, вы, скорее всего, получите высокоуровневую диаграмму с небольшим количеством коробок. Эта диаграмма будет основана на ментальной модели программного обеспечения разработчика. Однако, если вы реконструируете диаграмму из кодовой базы, вы получите совершенно иную картину. Он будет очень низкоуровневым, точным и точным, потому что инструменты реверс-инжиниринга обычно показывают вам отражение структур в коде.

Архитектурно очевидный стиль программирования

Ответ Джорджа на разрыв между моделью и кодом прост - мы должны использовать "архитектурно-очевидный стиль кодирования". Другими словами, мы должны сделать так, чтобы наш код отражал наши архитектурные идеи и намерения. В конкретном плане этого можно достичь путем:

- Соглашения об именовании: Если вы реализуете что-то, что вы считаете компонентом, убедитесь, что это очевидно из именования. Например, класс, интерфейс, пакет, пространство имен и т.д.) может включать слово "компонент".
- Соглашения об упаковке: Кроме того, возможно, вы группируете все, что связано с одним компонентом, в один пакет, пространство имен, модуль, папку и т.д.
- Машиночитаемые метаданные: В качестве альтернативы, почему бы не включить машиночитаемые метаданные в код, чтобы его части можно было проследить до архитектурного видения.
  В реальных терминах, например, вы можете использовать аннотации Java (например, `@Component`) или атрибуты C# (например, `[Component]`) для обозначения классов как архитектурно важных. Эти аннотации или атрибуты могут быть взяты из используемой вами платформы, или вы можете создать их самостоятельно.

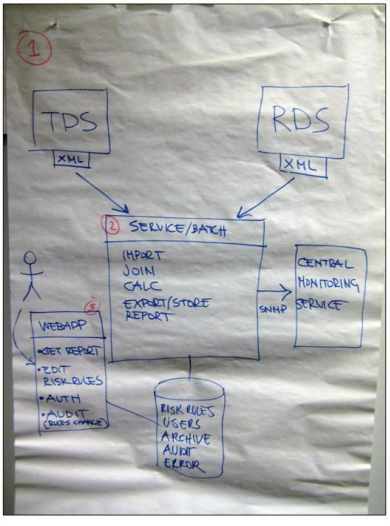
Проще говоря, если у вас на стене висит диаграмма компонентов, каждый из компонентов должен быть сопоставлен с чем-то реальным в коде. Если на схеме есть поле "Logging Component", убедитесь, что в коде действительно есть "Logging Component". Это относительно легко сделать, но, по моему опыту, я редко вижу, чтобы команды делали это. Наличие простого и явного сопоставления архитектуры с кодом может очень помочь, когда людей просят разобраться в кодовой базе, особенно если она для них новая. Кроме того, существует очевидная связь между архитектурой программной системы и тем, как эта архитектура визуализируется в виде набора диаграмм. Стиль архитектуры, который вы используете, должен быть отражен на диаграммах архитектуры вашего программного обеспечения; Будь то слои, порты и адаптеры, компоненты, микросервисы или что-то еще.

11.2 Детали технологии на диаграммах

Вспомните последнюю схему архитектуры программного обеспечения, которую вы видели. Как это выглядело? Какой уровень детализации он показал? Были ли включены варианты выбора технологий или они были исключены? По моему опыту, большинство диаграмм архитектуры программного обеспечения опускают какую-либо информацию о технологии, вместо этого сосредотачиваясь на иллюстрации функциональной декомпозиции и основных концептуальных элементов. Включение вариантов технологий (или опций) в диаграммы — еще один простой способ гарантировать, что диаграммы отражают реальность, и не позволяет им выглядеть как архитектура башни из слоновой кости, где куча концептуальных компонентов волшебным образом взаимодействует, образуя комплексную программную систему.

Построение схем в процессе проектирования

Вот фотография архитектурной схемы, созданной во время одного из моих семинаров по архитектуре программного обеспечения, и она довольно типична для того, что я вижу, из-за отсутствия каких-либо технологических деталей.



Если спросить людей, почему их диаграммы не показывают никаких технологических решений, это приведет к ряду различных ответов.

- «Решение [системы финансовых рисков] простое и может быть построено с использованием любой технологии».
- «Мы не хотим навязывать решение разработчикам».
- «Это деталь реализации».
- «Мы следуем принципу «последнего ответственного момента».

Построение схем ретроспективно

Если вы рисуете схемы архитектуры программного обеспечения ретроспективно, для документирования после того, как программное обеспечение было создано, на самом деле нет причин пропускать технологические решения. Тем не менее, другие не обязательно разделяют эту точку зрения, и я часто слышу следующие комментарии, когда спрашиваю людей, почему их схемы архитектуры программного обеспечения не включают технологии.

- «Технологические решения загромождают диаграммы».
- «Но все знают, что мы используем только базы данных ASP.NET и Oracle».

Сделайте выбор технологий очевидным

Кажется, что независимо от того, рисуются ли диаграммы до, во время или после создания программного обеспечения, существует распространенное заблуждение, что диаграммы архитектуры программного обеспечения не должны включать в себя технологию. Одна из причин того, что архитектура программного обеспечения имеет плохую репутацию, заключается в стереотипе о том, что архитекторы башен из слоновой кости рисуют очень высокоуровневые картины, чтобы описать свои грандиозные видения. Я уверен, что вы видели примеры диаграмм с большой коробкой с надписью «Enterprise Service Bus», подключенной к облаку, или, возможно, диаграммы, показывающие функциональную декомпозицию без какого-либо учета того, реализуема ли эта конструкция.

Мне нравится, когда диаграммы архитектуры программного обеспечения имеют под собой реальную основу, и я не считаю выбор технологии «деталью реализации». Один из способов обеспечить учет технологии — просто показать варианты технологий, включив их в диаграммы архитектуры программного обеспечения. Включение выбора технологий в схемы архитектуры программного обеспечения устраняет двусмысленность, даже если вы работаете в среде, где все программное

обеспечение создается с использованием стандартного набора технологий и шаблонов.

Конечно, если вы ретроспективно строите диаграмму существующей кодовой базы, технологические решения уже приняты, и поэтому у вас есть информация для добавления в схемы. Но как насчет того, когда схемы рисуются во время предварительного проектирования? Возможно, вы не знаете, какие технологии собираетесь использовать. Или, возможно, есть несколько вариантов.

Представьте, что вы разрабатываете программную систему. Вы действительно делаете это, не задумываясь о том, как вы на самом деле собираетесь это реализовать? Вы действительно мыслите в терминах логических строительных блоков и игнорируете технологии? Если ответ на эти вопросы «нет», то моя рекомендация — включить как можно больше информации. Например, если на схеме контейнера отображается база данных, но вы не уверены, будет ли это Microsoft SQL Server или MySQL, почему бы не указать это, добавив к элементу базы данных заметку "Microsoft SQL Server или MySQL". По крайней мере, это ясно дает понять, что решение должно быть принято, и показывает варианты, которые также рассматриваются. Это обеспечивает лучшую отправную точку для разговора, особенно если у вас есть выбор технологий для использования.

Поощрение людей к включению выбора технологий в свои схемы архитектуры программного обеспечения также имеет тенденцию приводить к гораздо более содержательным и глубоким разговорам, основанным на реальных событиях.
мир. Пушистая концептуальная диаграмма имеет тенденцию делать множество предположений, но учет технологий заставляет вместо этого задавать следующие типы вопросов.

- «Как этот компонент взаимодействует с этим компонентом, если он выполняется в отдельном процессе?»
- «Как инициируется этот компонент и на чем заключается его ответственность?»
- «Почему этот процесс должен взаимодействовать с этим процессом?»
- "почему этот компонент будет реализован в технологии X, а не в технологии Y?"
- и так далее

Выбор технологий может помочь вернуть идеалистический и концептуальный дизайн программного обеспечения на землю, чтобы он снова был основан на реальности, в то же время передавая всю общую картину, а не только ее часть.

Другим побочным эффектом добавления вариантов технологий на диаграммы, особенно в процессе разработки программного обеспечения, является то, что это помогает гарантировать, что их рисуют правильные люди (т.е. люди, которые разбираются в технологиях).

11.3 Вы бы закодировали это таким образом?

В качестве последнего замечания, связанного с созданием диаграмм, которые отражают реальность, если вы занимаетесь предварительным проектированием, простой вопрос, который следует задать себе: «Точно ли эта диаграмма отражает то, что мы собираемся создать? Будем ли мы кодировать это таким образом?». И если вы рисуете диаграммы для описания существующей кодовой базы, возникает вопрос: «Точно ли эта диаграмма отражает то, как мы создали программное обеспечение? Мы это так закодировали?».

Давайте проиллюстрируем это парой сценариев, которые я регулярно слышу на своих семинарах по архитектуре программного обеспечения.

Сценарий 1: Общие компоненты

Представьте, что вы разрабатываете программную систему, которая включает в себя два веб-приложения, одно из которых, возможно, предназначено для пользователей-людей, а другое предоставляет API. Когда мы думаем о компонентах, которые находятся в каждом из этих контейнеров, нередко можно услышать такой разговор:

- Участник: «Мы хотели бы использовать компонент общего ведения журнала в обоих веб-приложениях. Должны ли мы выводить этот компонент логирования за пределы двух контейнеров веб-приложений, поскольку он используется обоими?»
- Я: «Вы бы закодировали это таким образом? Будет ли компонент ведения журнала выполняться за пределами обоих веб-приложений? Например, действительно ли это будет отдельный контейнер или процесс?»
- Участник: «Нет, компонент ведения журнала будет общим компонентом в [JAR-файле|DLL|и т.д.], который мы развернем как часть обоих приложений».
- Я: «В таком случае, нарисуй его так. Включите компонент ведения журнала в каждое веб-приложение и просто пометьте его как общий компонент».

Если вы собираетесь реализовать что-то вроде общего компонента, который будет развернут в нескольких отдельных приложениях, убедитесь, что ваша схема отражает это, а не потенциально запутывает людей, включая что-то, что может быть ошибочно принято за отдельный централизованный сервер ведения журнала. Если вы сомневаетесь, всегда спрашивайте себя, как бы вы это написали.

Сценарий 2: Стратегии наслоения

Представьте, что вы разрабатываете веб-приложение, которое имеет внутреннюю структуру в виде уровня представления, уровня служб и уровня доступа к данным.

- Участник: «Должны ли мы показать, что все коммуникации с базой данных со стороны презентационного уровня проходят через уровень сервисов?»
- Я: «Вот как ты собираешься это реализовать? Или уровень представления будет обращаться к базе данных напрямую?»
- Участник: «Мы думали о возможном внедрении шаблона CQRS , чтобы для запросов на чтение уровень представления мог обойти уровень сервисов и напрямую использовать уровень доступа к данным».
- Я: «В таком случае нарисуйте диаграмму, как вы только что объяснили, с линиями от уровня представления к уровням сервисов и доступа к данным. Аннотируйте строки, чтобы указать на намерение и обоснование».

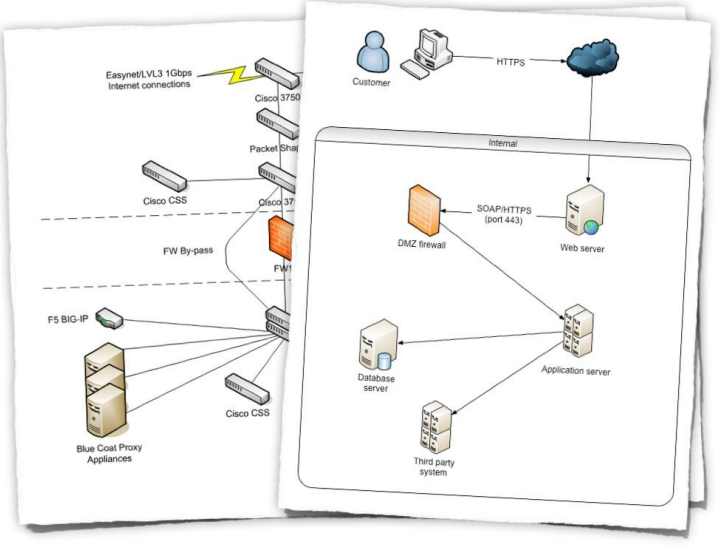
Опять же, самый простой способ ответить на этот вопрос — понять, как бы вы его написали. Если вы можете понять, как бы вы это закодировали, вы можете понять, как это визуализировать.

12. Схемы развертывания

Диаграммы моделей C4 предназначены для описания статической структуры программной системы на различных уровнях абстракции. Но для того, чтобы приносить пользу, это программное обеспечение должно быть выполнено где-то, в какой-то инфраструктуре. Раньше такой инфраструктурой были физические серверы, но сегодня это может быть что угодно — от виртуализированных серверов и «инфраструктуры как услуги» (IaaS) до контейнеров Docker и сред «платформа как услуга» (PaaS).

12.1 Схемы сети и инфраструктуры

В те времена, когда физические серверы были основным способом развертывания программного обеспечения в организациях, часто можно было увидеть, как команды создают «сетевые диаграммы» или «инфраструктурные схемы». По сути, это была карта сетевой инфраструктуры; отображение серверов, маршрутизаторов, межсетевых экранов, балансировщиков нагрузки, сетевых коммутаторов, виртуальных локальных сетей и т. д.



Примеры схем инфраструктуры, созданных вручную в Microsoft Visio

Эти диаграммы, как правило, создавались вручную инженерами по инфраструктуре, сетям и эксплуатации, и часто не очень хорошо понимались инженерами-программистами. Эти диаграммы определенно полезны, но обычно не показывают подробностей о том, где и как будет работать программное обеспечение. Кроме того, команда разработчиков обычно располагала набором диаграмм инфраструктуры, а команда разработчиков — отдельным набором диаграмм архитектуры программного обеспечения, что приводило к разрыву между этими двумя представлениями.

12.2 Схемы развертывания

Движение DevOps стремилось объединить эти две отдельные дисциплины. С точки зрения построения диаграмм, один из способов сделать это — описать, как программное обеспечение развертывается в инфраструктуре на «диаграмме развертывания». Даже если развертывание полностью автоматизировано, все равно может быть полезно иметь схему, обобщающую сопоставление развертывания. Например, веб-сайт, управляемый базой данных, может быть развернут на одном сервере или на ферме серверов, состоящей из сотен серверов, в зависимости от необходимости поддержки масштабируемости, устойчивости, безопасности и т. д.

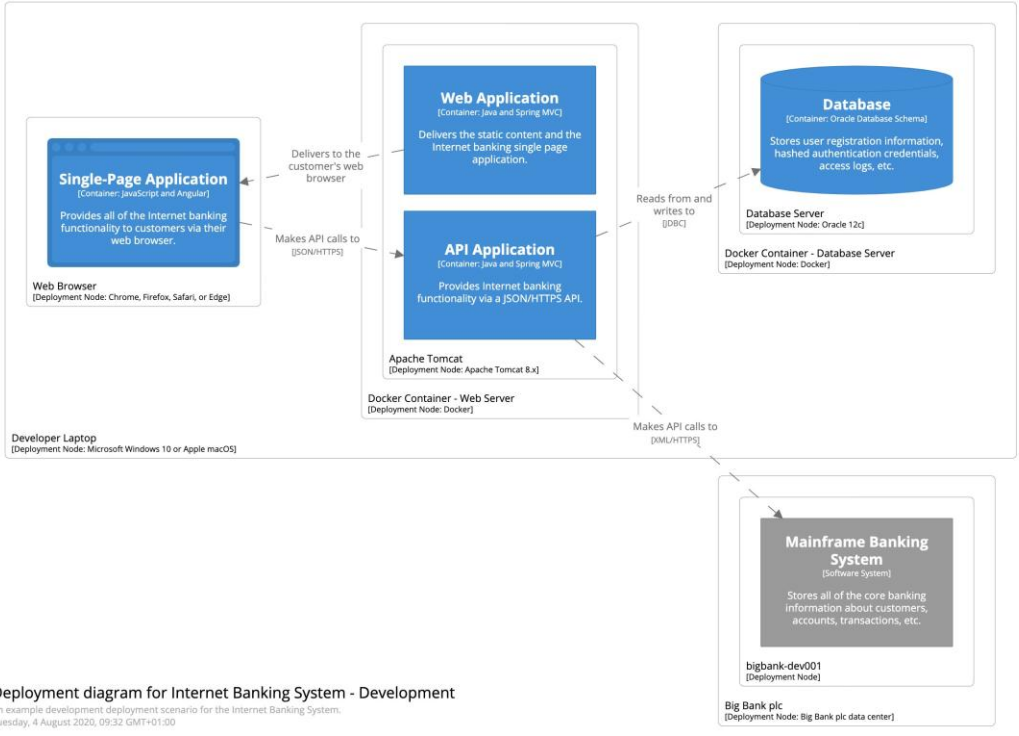
Концепция диаграммы развертывания пришла из UML и используется для описания отображения артефактов развертывания (например, развертываемой единицы, такой как JAR-файл) на узлы развертывания (т.е. устройства или среды выполнения). Диаграммы развертывания UML действительно работают, но я часто обнаруживаю, что они кажутся «оторванными» от других представлений программного обеспечения. Например, диаграмма развертывания UML может отображать определенный артефакт (например, JAR-файл или библиотеку DLL), развернутый на сервере, но часто нет другой диаграммы, определяющей, что содержится в этом развертываемом артефакте.

Взяв за основу модель C4, я использую несколько более простой подход, в соответствии с которым мои диаграммы развертывания показывают отображение программных систем или контейнеров (из модели C4) на узлы развертывания. Узел развертывания — это что-то вроде:

- Физическая инфраструктура (например, физический сервер или устройство).

- Виртуализированная инфраструктура (например, IaaS, PaaS, виртуальная машина).
- Контейнерная инфраструктура (например, контейнер Docker).
- Среда выполнения (например, сервер базы данных, веб-сервер/сервер приложений Java EE, Microsoft IIS и т. д.).

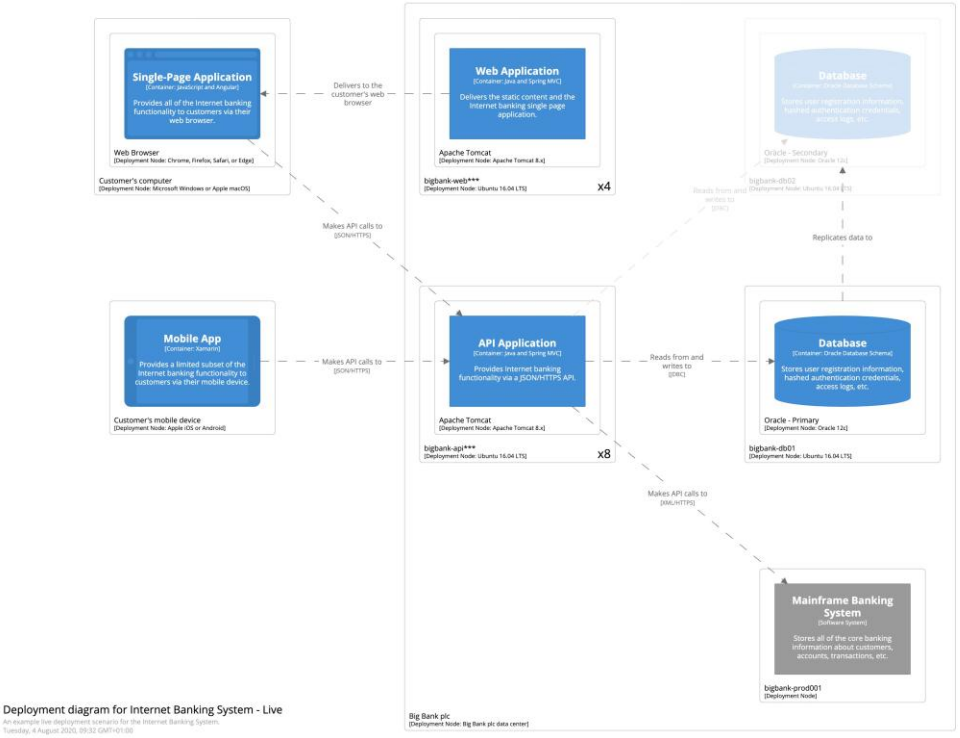
Кроме того, узлы развертывания могут быть вложенными. По сути, вы показываете экземпляры программных систем/контейнеров и то, как они развертываются в инфраструктуре. В качестве примера на следующей схеме показано, как может выглядеть среда разработки для системы интернет-банкинга.



На ноутбуке разработчика установлен собственный веб-браузер, и именно там находится Single Page
Приложение будет работать, когда мы будем заниматься разработкой. Затем есть пара контейнеров Docker. Один из них работает под управлением копии Apache Tomcat, в котором мы развертываем разрабатываемые версии веб-приложений и приложений

API. Другой контейнер Docker работает с нашей базой данных разработки. Мы также используем разрабатываемую версию банковской системы Mainframe, развернутую на сервере разработки где-то в банке.

В качестве немного более сложного примера давайте рассмотрим, как может выглядеть динамическая среда развертывания.

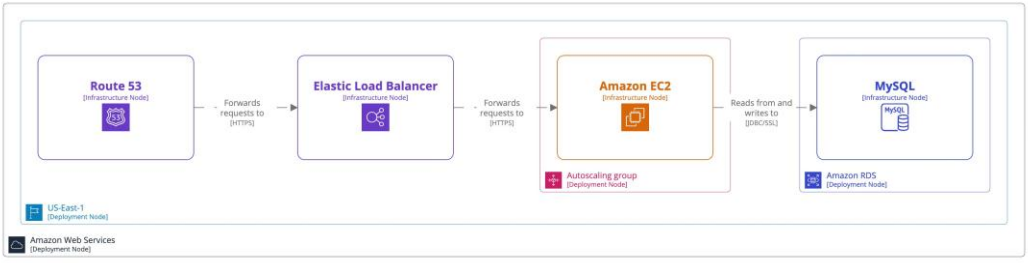


У клиентов могут быть компьютеры или мобильные устройства, на которых в конечном итоге и работают пользовательские интерфейсы. Все остальное развернуто в дата-центре Big Bank, на нескольких отдельных серверах Ubuntu. Я также показал вторичный сервер базы данных с репликацией данных.

Вы также можете включить «узлы инфраструктуры», такие как службы DNS, маршрутизаторы, балансировщики нагрузки, брандмауэры и т. д. Не стесняйтесь также указывать имена хостов, IP-адреса или любую другую информацию, которая, по вашему мнению, поможет вам рассказать историю, которую вы хотите рассказать.

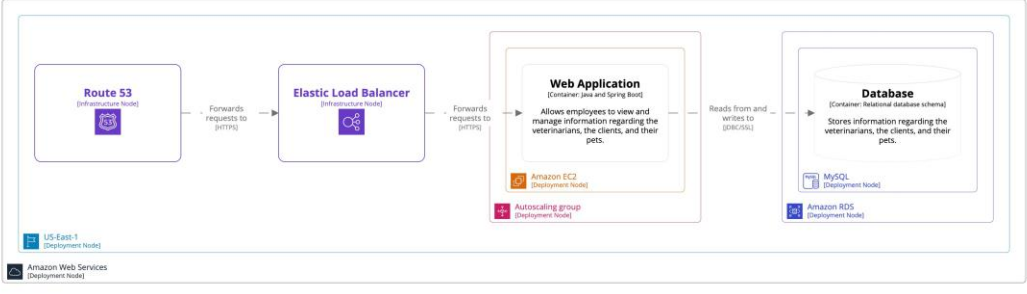
12.3 Схемы архитектуры облака

Я вижу много дискуссий о «диаграммах облачной архитектуры», люди конкретно спрашивают, как их создавать, какие инструменты использовать и где взять наборы иконок. Многие из примеров диаграмм облачной архитектуры, которые я вижу, относительно универсальны и выглядят примерно так.



На этой схеме показано, что используется ряд сервисов AWS. включая Route 53, Elastic Load Balancer, EC2 (с автоматическим масштабированием) и MySQL на RDS. Он хорошо выглядит, он красочный и использует соответствующие иконки из набора иконок AWS. Это неплохая первая попытка, потому что она помогает определить, какие сервисы AWS используются. Но это также самая большая проблема - это действительно диаграмма инфраструктуры, и она рассказывает вам только половину истории. Понимание того, какие сервисы AWS используются, – это здорово, но мне также хотелось бы знать, как вы их используете.

Большинству команд разработчиков программного обеспечения было бы лучше создать схему развертывания, которая дополнительно показывает отображение контейнеров (приложений и хранилищ данных) на эту инфраструктуру. Вот пример получше.



На этой диаграмме снова показаны используемые сервисы AWS, но также показано, для чего они используются, включая веб-приложение и базу данных. Конечно, вы можете добавить больше деталей по мере необходимости; включая VPC, VLAN, межсетевые экраны, IP-адреса, количество экземпляров, ограничения масштабирования и т. д. Стоит отметить, что тот же совет относится и к построению схем сред Docker и Kubernetes — не сосредотачивайтесь только на инфраструктуре, не забывайте показывать свои собственные приложения и хранилища данных.

13. Другие диаграммы

Контекстные, контейнерные и компонентные диаграммы, которые мы видели до сих пор, часто достаточны для описания программной системы. Тем не менее, иногда может быть полезно нарисовать несколько дополнительных диаграмм, чтобы выделить различные аспекты.

13.1 Архитектурные модели

Существует несколько различных способов осмысления, описания и визуализации программной системы. Примеры включают IEEE 1471, ISO/IEC/IEEE 42010, модель 4+1 Филиппа Крухтена и т. д. Общим для них является то, что все они предоставляют различные «взгляды» на программную систему для описания различных ее аспектов. Например, часто существует «логическое представление», «физическое представление», «представление разработки» и так далее.

Возможно, вас интересует, как модель C4 соотносится с некоторыми из более популярных архитектурных моделей видов, такими как архитектурная модель 4+1 Филиппа Крухтена и коллекция, предлагаемая в книге «Архитектура программных систем » Эоина Вудса и Ника Розански.

Посмотреть модель "4+1" (Филипп Крухтен)

Модель представления «4+1» состоит из пяти различных представлений, которые могут быть использованы для описания программной системы. Оригинальное определение можно найти в статье Филиппа IEEE "Architectural Blueprints - The "4+1" View Model of Software Architecture, хотя многие люди усовершенствовали эту модель с момента публикации оригинальной статьи. Кое-что из того, что вы найдете в Интернете о «4+1», не обязательно совпадает с содержанием оригинальной статьи, и многие люди тонко пересмотрели представления модели, чтобы лучше сопоставить их с нотацией или методологией, которую они использовали в то время (например, UML). Мое резюме "4+1" выглядит следующим образом:

- Логическое представление: Это представление описывает функциональные возможности, предоставляемые системой. Обычно это одна или несколько

высокоуровневых диаграмм, которые показывают основные функциональные строительные блоки и их взаимосвязь.
- Представление процесса: Это представление описывает, как логические строительные блоки объединяются в физические процессы (или исполнительные единицы). Он используется для захвата параллелизма, межпроцессного взаимодействия и т. д.
- Физическое представление: Это представление описывает инфраструктуру и топологию развертывания программного обеспечения.
- Представление разработки: Это представление описывает, как функциональные строительные блоки в логическом представлении реализуются разработчиками программного обеспечения с помощью модулей, компонентов, классов, слоев и т. д.
- Представление сценариев: ряд выбранных вариантов использования или сценариев использования используется для управления и иллюстрации содержимого четырех предыдущих представлений.

Архитектура программных систем (Эоин Вудс и Ник Розански)

Книга «Архитектура программных систем» Эоина Вудса и Ника Розански определяет еще одну модель описания программной системы. Она основана на модели представления «4+1» и представляет собой коллекцию из семи точек обзора, а именно:

- Контекстная точка зрения: Описывает, как программная система вписывается в окружающую среду (т.е. людей и другие программные системы).
- Функциональная точка зрения: Это похоже на "Логическое представление" в модели "4+1"; В нем показаны функциональные строительные блоки, составляющие программную систему. Она была переименована, чтобы сделать намерение более понятным (т.е. вы смотрите на функциональные строительные блоки, составляющие программную систему).
- Информационная точка зрения: Это описание того, как программная система хранит и использует информацию, с точки зрения статической структуры (например, модели отношений между сущностями и т.д.), как эта информация используется во время выполнения (например, поток информации через систему, модели информационного состояния и т.д.), как она архивируется, объемность и т.д. Такая точка зрения позволяет рассматривать информацию как

гражданина первого сорта, а не как побочный продукт системы программного обеспечения.
- Точка зрения параллелизма: Это похоже на "Представление процесса" в модели "4+1".
- Точка зрения развития: Это похоже на "Вид разработки" в модели "4+1".
- Точка зрения развертывания: аналогична "физическому виду" в модели "4+1".
- Операционная точка зрения: Эта точка зрения используется для описания эксплуатационных аспектов программного обеспечения; включая установку, эксплуатацию, модернизацию, миграцию данных, управление конфигурацией, администрирование, мониторинг, поддержку моделей и т. д.

Общий словарный запас

Большая проблема, которую я обнаружил в реальном мире со многими из этих существующих подходов, заключается в том, что они очень быстро начинают запутываться, если вся команда не разбирается в используемой терминологии. Например, я слышал, как люди спорят о том, в чем разница между «концептуальным взглядом» и «логическим взглядом». И давайте даже не будем задавать вопросы о том, разрешены ли технологии в логическом плане. Перспектива тоже важна. Если я разработчик программного обеспечения, то это "представление разработки" о коде или это "представление реализации"? Но как насчет «физического вида»? В конце концов, код — это физический вывод. Но тогда «физический вид» означает нечто иное для архитекторов инфраструктуры. Но что делать, если целевая среда развертывания является виртуальной, а не физической?

Общей темой всей этой книги было создание общего словаря, и то же самое применимо, когда вы думаете о том, какие еще диаграммы нарисовать. Один из способов решения проблемы с терминологией заключается в том, чтобы убедиться, что каждый член команды может указать на четкое определение различных типов диаграмм или архитектурных представлений. Просто имейте в виду, что в разных книгах по архитектуре программного обеспечения часто используются разные названия для описания одного и того же архитектурного представления. Этот включал, конечно.

Преодоление разрыва между моделью и кодом

По моему опыту, «логическое представление» (или «функциональное представление») — это обычно то, о чем люди думают, когда их просят нарисовать «архитектурную диаграмму». Несмотря на то, что это полезное представление для

создания, оно часто сбивает с толку, особенно для разработчиков программного обеспечения, когда это логическое представление функциональных строительных блоков не так легко сопоставить с кодом или отразить его. Кроме того, это разделение между «логическим взглядом» и «взглядом на развитие» я часто вижу в организациях, где существует четкое разделение между «архитекторами» и «разработчиками». Это не обязательно плохо, но переход между «логическим» и «разработочным» представлениями не обязывает передавать процессы между отдельными командами архитектуры и разработчиков. Передача процессов имеет тенденцию еще больше преувеличивать расстояние между «логическим» и «развивающим» подходами.

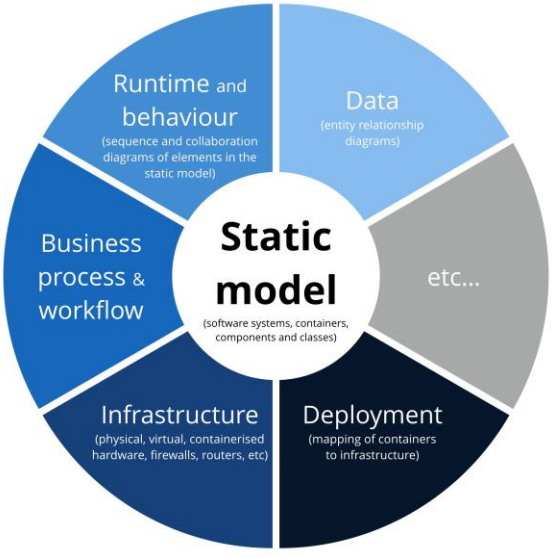
Лично я предпочитаю свести к минимуму и, по сути, устранить разрыв между логическим представлением о том, что делает система, и реальным взглядом на то, как эта функциональность реализована в коде. Как мы уже обсуждали в предыдущих главах, речь идет о минимизации разрыва в модельном коде. По сути, модель C4 охватывает и объединяет то, что вы можете найти в «логическом представлении» и «представлении разработки», в единое описание статической структуры на нескольких различных уровнях абстракции:

- Системный контекст: Он показывает вам, что делает система и как она вписывается в окружающий мир (т.е. пользователей и другие программные системы).
- Контейнеры: Здесь показано, как функциональные возможности, предоставляемые системой, распределяются по высокоуровневым строительным блокам (т.е. контейнерам).
- Компоненты: Здесь показано, как функциональные возможности, предоставляемые определенным контейнером, распределяются между компонентами внутри этого контейнера.

Можно сказать, что модель C4 также включает в себя некоторые из того, что вы найдете в «Процессе 4+1»
Вид», особенно с учетом того, что на схеме контейнера отображаются единицы выполнения. Это, безусловно, так, хотя все еще могут быть случаи, когда стоит создать специализированную версию диаграммы контейнера, чтобы подчеркнуть параллелизм или синхронизацию.

Сначала разберитесь со статической структурой

Основное внимание в этой книге уделяется описанию и передаче статической структуры программной системы; От общей картины того, как программная система вписывается в свою среду, до ее компонентов и классов, которые их реализуют. Как только у вас есть общий словарь, с помощью которого можно описать статическую структуру программной системы (на различных уровнях абстракции), становится легко сообщать о других аспектах этой системы, основанных на статической структуре.



Статическая структура определяет ядро модели архитектуры программного обеспечения

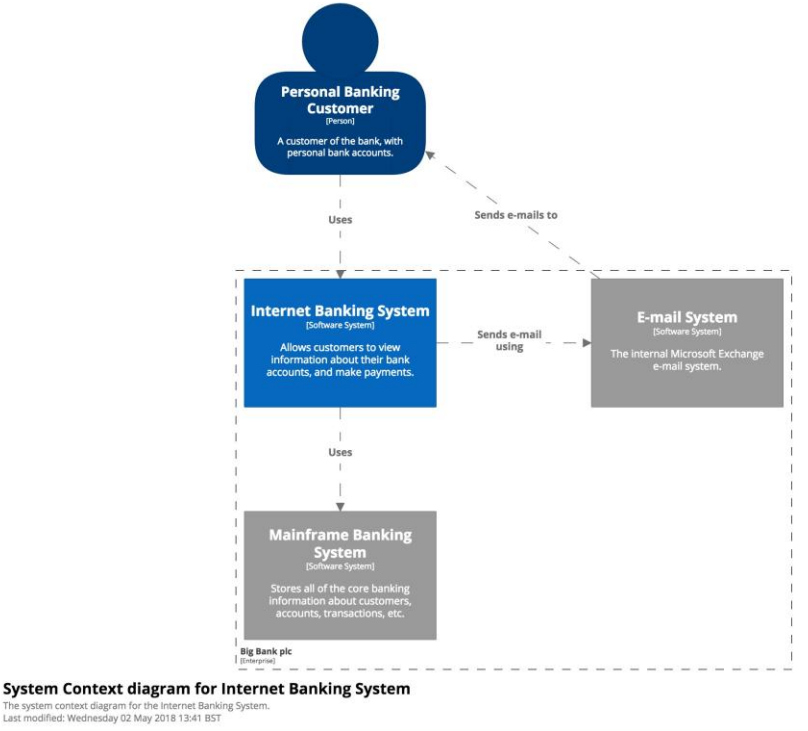
Имея это в виду, вот некоторые другие диаграммы, которые вы, возможно, захотите создать.

13.2 Системный ландшафт

Модель C4 обеспечивает статическое представление одной программной системы, но в реальном мире программные системы никогда не живут изолированно. По этой причине, особенно если вы отвечаете за набор программных систем, часто бывает полезно понять, как все эти программные системы вписываются друг в друга в рамках предприятия. Для этого я добавлю еще одну диаграмму, которая будет располагаться «сверху» диаграмм C4, чтобы показать системный ландшафт с точки

зрения ИТ. Как и диаграмма системного контекста, эта диаграмма может отображать организационные границы, внутренних/внешних пользователей и внутренние/внешние системы.

С практической точки зрения, диаграмма системного ландшафта на самом деле является просто диаграммой системного контекста без особого фокуса на конкретной программной системе. Вот пример диаграммы системного контекста для системы интернет-банкинга, которую мы видели ранее.



Системная контекстная диаграмма для вымышленной системы интернет-банкинга

При составлении диаграммы системного контекста я обычно включаю только тех людей и программные системы, которые имеют непосредственное отношение к рассматриваемой программной системе. В отличие от этого, диаграмма системного ландшафта обычно показывает большую часть ИТ-ландшафта, чтобы рассказать более широкую общую картину.

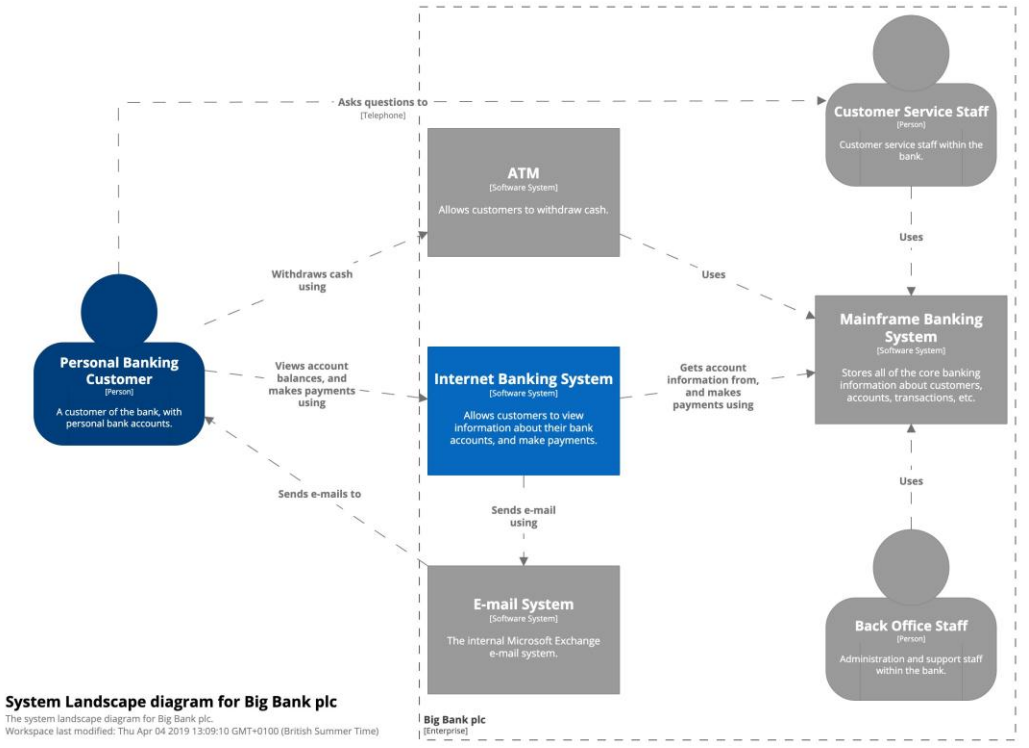
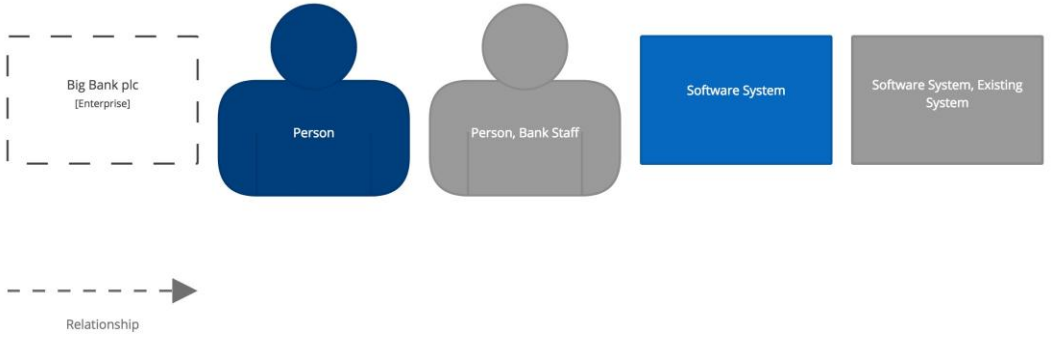


Диаграмма системного ландшафта для вымышленного "Big Bank plc"

Опять же, я использовал прямоугольник с пунктирной рамкой для представления границы предприятия, наряду с некоторыми дополнительными стилями элементов, чтобы мы могли видеть, что является внутренним для предприятия, а что внешним.



Ключ к диаграмме системного ландшафта

В качестве предостережения, я понимаю, что корпоративная архитектура — это не просто технология, но, по моему опыту, многие организации не имеют представления о корпоративной архитектуре своего ИТ-ландшафта. На самом деле, меня шокирует, как часто я вижу организации всех размеров, которым не хватает такого целостного взгляда, особенно если учесть, что технологии обычно являются ключевой частью того, как они реализуют бизнес-процессы и обслуживают клиентов. Диаграмма предприятия с технологической точки зрения, по крайней мере, дает возможность выйти за рамки типичных разрозненностей, которые формируются вокруг ИТ-систем и команд, которые за них отвечают.

13.3 Макеты и каркасы пользовательского интерфейса

Создание макетов пользовательских интерфейсов с помощью таких инструментов, как Balsamiq , — это фантастический способ понять, что требуется от программной системы и создать прототип идей. Такие эскизы, макеты и каркасы дадут представление о программном комплексе, которое невозможно получить с моделью C4.

13.4 Бизнес-процессы и документооборот

С диаграммами последовательностей и совместной работы связаны модели процессов. Иногда мне хочется обобщить конкретный бизнес-процесс или рабочий процесс пользователя, который реализует программная система, не вдаваясь в технические детали его реализации. Диаграмма активности UML или традиционная блок-схема — отличный способ сделать это.

13.5 Модель предметной области

Большинство реальных программных систем представляют собой нетривиальные бизнес-области, и в этом случае диаграмма, обобщающая ключевые понятия предметной области, может быть полезным дополнением. Я использую для этого диаграмму классов UML, где каждый класс UML представляет сущность в предметной области. Для каждой сущности я обычно включаю важные атрибуты/свойства и отношения между сущностями. Модель предметной области

полезна независимо от того, следуете ли вы Предметно-ориентированный дизайн подходить или нет.

13.6 Время выполнения и поведение

Большая часть этой книги сосредоточена на подходе к осмыслению, описанию и передаче архитектуры программного обеспечения, который сосредоточен вокруг статической структуры: программных систем, контейнеров, компонентов и классов. Всякий раз, когда мне приходилось документировать программную систему в прошлом, во всяком случае, с точки зрения построения диаграмм, большая часть того, что я создавал, перекликалась с этим чувством, причем большинство моих диаграмм были описаниями статической структуры. Мои мастер-классы по созданию эскизов архитектуры программного обеспечения также подтверждают аналогичную тенденцию. Во время начальной итерации (где я даю очень мало указаний и, надеюсь, это не повлияет на результат) я оцениваю, что 80-90% диаграмм, которые я вижу, отражают статическую структуру.

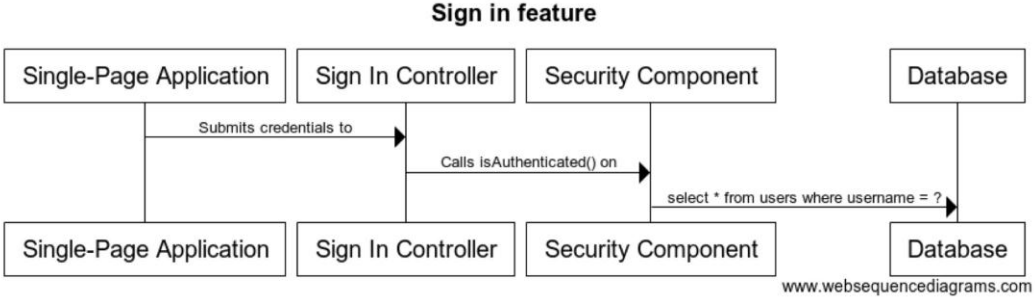
Однако программное обеспечение не статично, и его нужно выполнять, чтобы действительно делать что-то полезное. По этой причине может быть полезно создавать схемы, иллюстрирующие то, что происходит во время выполнения («динамическое представление») для важных сценариев использования, пользовательских историй или сценариев.

Для этого я просто беру концепцию последовательностей и диаграмм взаимодействия/коммуникации из UML и применяю их к статическим элементам в модели C4. Например, можно проиллюстрировать, как реализуется вариант использования, нарисовав диаграмму последовательности взаимодействия компонентов во время выполнения. Или вы можете показать взаимодействие между контейнерами, если у вас больше микросервисный стиль архитектуры, где каждый сервис развертывается в отдельном контейнере.

UML-диаграммы последовательностей и совместной работы, как правило, являются способом, которым это делает большинство людей, включая меня, хотя я склонен использовать UML в качестве обозначения для набросков, а не точно следовать спецификации. В Интернете есть много введений в UML, но, по сути, оба типа диаграмм показывают одну и ту же информацию, хотя и с несколько иной точки зрения.

Диаграммы последовательностей

Диаграмма последовательности UML обычно состоит из ряда элементов (слева направо) и временной шкалы (сверху вниз). На схеме показано, как различные элементы взаимодействуют друг с другом (с помощью горизонтальных стрелок) при отправке сообщений, запросах и т. д. Вертикальный порядок стрелок иллюстрирует последовательность. Диаграммы последовательностей можно использовать для иллюстрации любой последовательности взаимодействующих элементов. Обычно эти элементы представляют собой элементы кода, такие как классы, но ничто не мешает вам показать людей, программные системы, контейнеры или компоненты. В качестве примера приведем схему последовательности, иллюстрирующую, как работает процесс входа в систему интернет-банкинга.

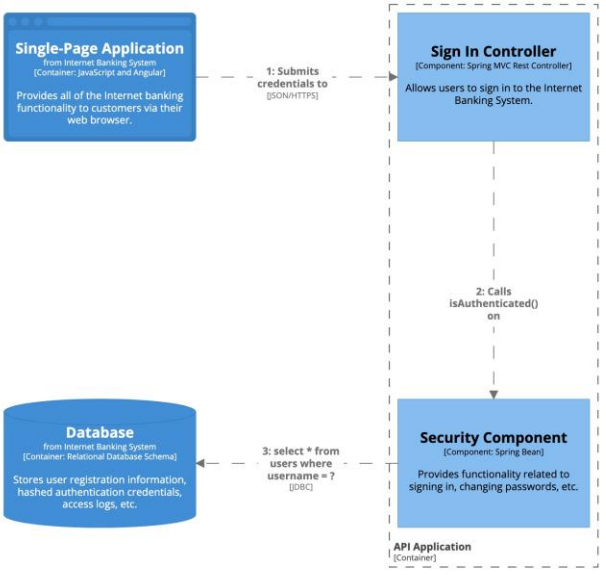


Существует ряд инструментов и подходов к созданию диаграмм последовательностей, но все они создают похожие диаграммы, при этом в официальной спецификации UML подробно описаны способы повышения точности и семантики диаграмм.

Сотрудничество

Другой подход — это диаграмма сотрудничества[1]. По сути, она показывает ту же информацию, что и диаграмма последовательностей, хотя эта информация представлена в другом виде. Как правило, это более простая диаграмма в стиле «коробки и линии», в которой линии аннотированы и пронумерованы, чтобы указать порядок совместной работы. Ниже приведен тот же сценарий входа, на этот раз проиллюстрированный с помощью диаграммы совместной работы.

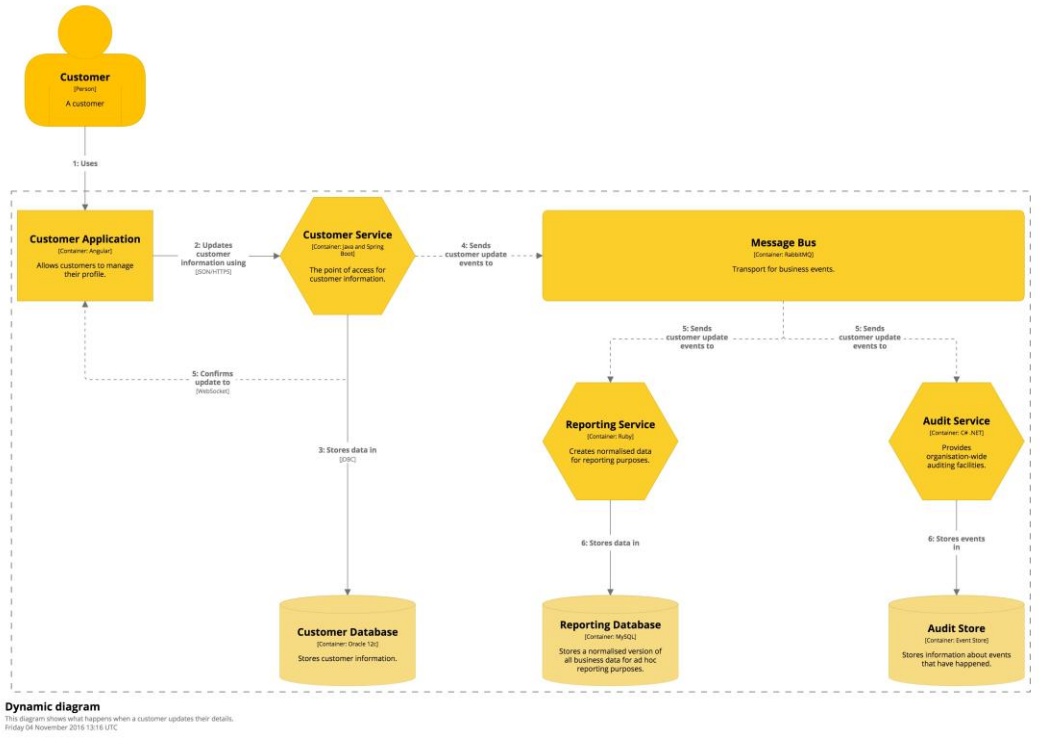
[1] В UML 2.x это называется коммуникационной диаграммой, но цель и содержание по сути те же.



Dynamic diagram for API Application

Summarises how the sign in feature works in the single-page application.
Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

Вы также можете использовать нумерацию строк для иллюстрации параллелизма, просто присвоив нескольким строкам один и тот же номер.



В этом примере архитектуры микрослужб шина сообщений транслирует сообщение "событие обновления клиента" в Службу отчетности и Службу аудита одновременно, поэтому я пронумеровал эти строки одинаково ("5"). Опять же, в зависимости от истории, которую вы хотите рассказать, вы можете выбрать, насколько точным вы будете. Службы отчетности и аудита хранят данные в хранилище данных, и я решил пронумеровать обе эти строки одинаково («6»), предполагая, что это происходит параллельно. Но, конечно, это может быть так, а может и не быть, в зависимости от того, сколько времени требуется каждому сервису для обработки события. В большинстве случаев этого достаточно, чтобы рассказать историю. Если вы хотите быть более точным, вы можете использовать концепцию вложенной последовательности UML, используя "5.1" вместо "6". Лично меня сбивает с толку использование вложенной нумерации последовательностей, особенно когда вложенность начинает углубляться более чем на пару уровней, поэтому я стараюсь избегать этого. Но это, безусловно, еще один инструмент в наборе инструментов.

Я предпочитаю диаграмму совместной работы, потому что ее легче рисовать, особенно на доске, но любой тип диаграммы работает, если вы не пытаетесь показать

слишком много совместных работ. И не забывайте все советы по нотации, о которых вы читали ранее, потому что они в равной степени применимы и здесь.

Когда следует создавать динамические виды

Учитывая количество путей выполнения даже через относительно небольшую программную систему
(пользовательские истории, сценарии успеха/неудачи, условная логика, асинхронная обработка и т. д.), очевидно, нецелесообразно документировать все. Особенно на уровне класса или метода. На самом деле, если вы хотите выяснить, как что-то работает, часто проще просто погрузиться в код, запустить автоматический тест или использовать отладчик. Конечно, это предполагает, что у вас есть хорошая отправная точка и вы знаете, где искать.

Если я вспомню о программных системах, которые я документировал, и о моем подходе к документированию в целом, я очень редко описываю динамические аспекты программной системы. Вот несколько примеров, где я это делал:

- Объяснение того, как низкоуровневый шаблон проектирования работает на уровне кода (взаимодействия между классами).
- Объяснение взаимодействия между приложениями и службами во время проверки подлинности пользователей при использовании федеративного поставщика безопасности (например, установление связи и взаимодействие между ASP.NET приложением, работающим под управлением Windows Identity Foundation, и службами федерации Microsoft Active Directory не является простым процессом).
- Объяснение типичного потока асинхронных сообщений/событий, реализующих бизнес-процесс.

Хотя динамические аспекты программной системы, безусловно, важны, я обычно не считаю, что их документирование приносит большую пользу. Как я уже сказал, вместо того, чтобы документировать каждый путь выполнения через программную систему, я буду делать это только для того, чтобы объяснить значимые или сложные сценарии, особенно когда они не очевидны при чтении кода.

13.7 И многое другое

Диаграмм модели C4 плюс те, которые я перечислил здесь, обычно достаточно для того, чтобы адекватно описать, как проектируется, строится и работает программная система. Я стараюсь свести к минимуму количество диаграмм, которые я использую для этого, и советую вам делать то же самое. Некоторые диаграммы могут быть сгенерированы автоматически (например, диаграмма отношений сущностей для схемы базы данных), но если для ее отображения требуется лист бумаги формата A0, следует подумать о том, действительно ли эта диаграмма полезна. Создавайте больше диаграмм, если вам нужно описать что-то, чего здесь нет, и если конкретная диаграмма не приносит никакой ценности, просто отбросьте ее.

14. Приложение A: Система финансовых рисков

14.1 Предыстория

Глобальный инвестиционный банк, базирующийся в Лондоне, Нью-Йорке и Сингапуре, торгует (покупает и продает) финансовыми продуктами с другими банками (контрагентами). Когда цены на акции на фондовых рынках движутся вверх или вниз, банк либо зарабатывает деньги, либо теряет их. В конце рабочего дня банку необходимо получить представление о том, какому риску он подвержен (например, потере денег), проведя некоторые расчеты на основе имеющихся у него данных о сделках. У банка уже есть Система торговых данных (TDS) и Система справочных данных (RDS), но ему нужна новая система управления рисками.

Система торговой информации

Система торговых данных хранит все сделки, совершенные банком. Он уже настроен на генерацию экспорта торговых данных в формате XML на основе файлов в конце рабочего дня (17:00) в Нью-Йорке. Экспорт включает в себя следующую информацию по каждой сделке, совершенной банком:

- Идентификатор сделки
- Дата
- Текущая стоимость торговли в долларах США
- Идентификатор контрагента

Система нормативно-справочной информации

Система справочных данных поддерживает все необходимые банку справочные данные. Сюда входит информация о контрагентах; Каждый из которых представляет физическое лицо, банк и т.д. Также доступен экспорт в формате XML на основе файлов, который включает основную информацию о каждом контрагенте. В ближайшие 3 месяца должна быть завершена работа над новой системой справочных

данных в масштабах всей организации, при этом существующая система в конечном итоге будет выведена из эксплуатации.

14.2 Функциональные требования

Высокоуровневые функциональные требования к новой Системе управления рисками заключаются в следующем.

А. Импортируйте торговые данные из Системы торговых данных.
Б. Импорт данных контрагента из Системы справочных данных.
В. Объедините два набора данных вместе, обогатив торговые данные информацией о контрагенте.
Г. Для каждого контрагента рассчитайте риск, которому подвержен банк.
Д. Создание отчета, который может быть импортирован в Microsoft Excel и содержит известные банку показатели рисков по всем контрагентам.
Е. Распространите отчет среди бизнес-пользователей до начала следующего торгового дня (9 утра) в Сингапуре.
Ж. Предоставление подмножеству бизнес-пользователей возможности настраивать и поддерживать внешние параметры, используемые при расчете рисков.

14.3 Нефункциональные требования

Нефункциональные требования к новой Системе управления рисками заключаются в следующем.

Производительность

- Отчеты о рисках должны быть составлены до 9 утра следующего рабочего дня в Сингапуре.

Масштабируемость

- Система должна быть способна справляться с объемами торговли в течение следующих 5 лет.
- Экспорт Системы торговых данных в настоящее время включает около 5000 сделок, и ожидается, что будет еще 10 сделок в день.

- Экспорт контрагентов ССИ включает около 20 000 контрагентов, и рост будет незначительным.
- В мире насчитывается 40-50 бизнес-пользователей, которым необходим доступ к отчету.

Наличие

- Отчеты о рисках должны быть доступны пользователям 24 часа в сутки 7 дней в неделю, но небольшое время простоя (менее 30 минут в день) может быть допустимо.

Отработка отказа

- Переключение на другой ресурс вручную достаточно для всех компонентов системы при условии достижения целевых показателей доступности.

Безопасность

- Эта система должна следовать политике банка, согласно которой доступ к системе разрешен только аутентифицированным и авторизованным пользователям.
- Отчеты должны распространяться только среди авторизованных пользователей.
- Только подгруппа авторизованных пользователей имеет право изменять параметры, используемые при расчете рисков.
- Хотя это желательно, нет требований к единому входу (например, интеграция с Active Directory, LDAP и т. д.).
- Весь доступ к системе и отчетам будет осуществляться в пределах глобальной сети банка.

Ревизия

- В журналах аудита системы должны быть зафиксированы следующие события: — Генерация отчета.
  — Изменение параметров расчета риска.
- Должна быть возможность понять исходные данные, которые использовались при расчете риска.

Отказоустойчивость и отказоустойчивость

- Система должна предпринять соответствующие шаги для восстановления после ошибки, если это возможно, но все ошибки должны быть зарегистрированы.
- Ошибки, препятствующие завершению расчета риска контрагента, должны быть зарегистрированы, и процесс должен быть продолжен.

Интернационализация и локализация

- Все пользовательские интерфейсы будут представлены только на английском языке.
- Все доклады будут представлены только на английском языке.
- Все торговые показатели и показатели риска будут представлены только в долларах США.

Мониторинг и управление

- Ловушка протокола SNMP должна быть отправлена в Центральную службу мониторинга банка в следующих случаях:
  - Когда происходит фатальная ошибка с системным компонентом.
  - Если отчеты не были сформированы до 9 утра по сингапурскому времени.

Хранение и архивирование данных

- Входные файлы, используемые в процессе расчета рисков, должны храниться в течение 1 года.

Функциональная совместимость

- Интерфейсы с существующими системами данных должны соответствовать существующим форматам данных и использовать их.